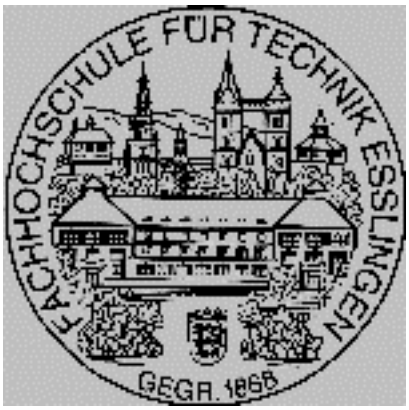
Fachhochschule Esslingen

Hochschule für Technik

Fachbereich Informationstechnik

Prof. Dr. rer. nat. Heinrich Weber

Prof. Dr. rer. nat. Manfred Dausmann



Skriptum zur Vorlesung

Grundlagen der Informatik 2

(Software-Engineering)

WS 2002/2003

Fachhochschule Esslingen - Hochschule für Technik

Fachbereich Informationstechnik

Flandernstrafle 101

73732 Esslingen a. N.

Inhaltsverzeichnis

1 Einführung

2 Programmentwicklungszyklus

3 Problemanalyse

4 Strukturierte Analyse

5 Die Methode SA/RT

6 Strukturierter Entwurf

7 Implementierung

8 Überprüfung und Test

9 Phasenübergreifende Tätigkeiten

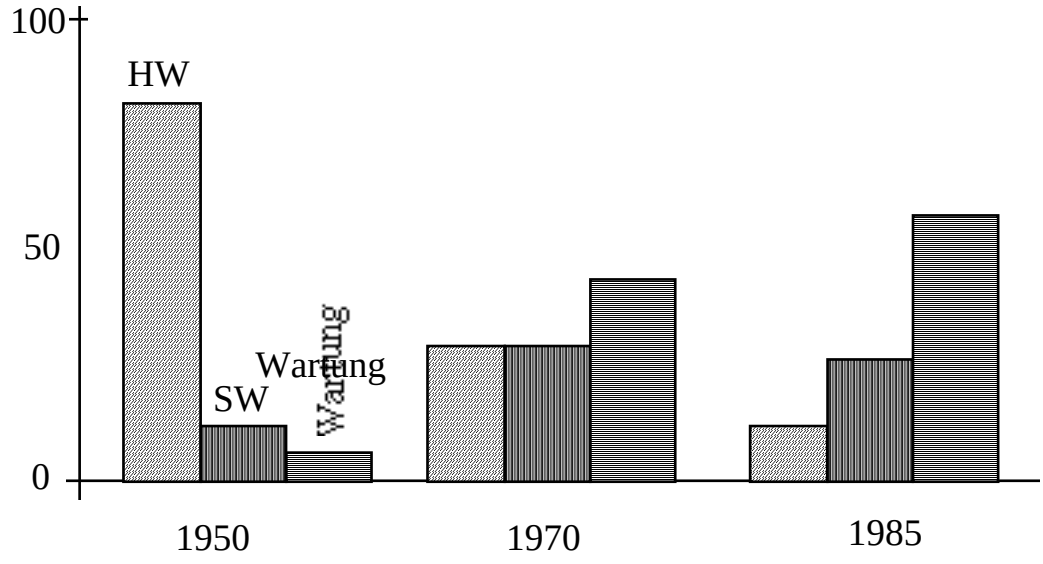
10 Literaturverzeichnis

1 Einführung

In den 60-er Jahren verursachten die Kosten für die Hardware den größten Teil der EDV-Kosten. Die Softwarekosten waren vernachlässigbar gering.

Im Laufe der Zeit änderte sich dies:

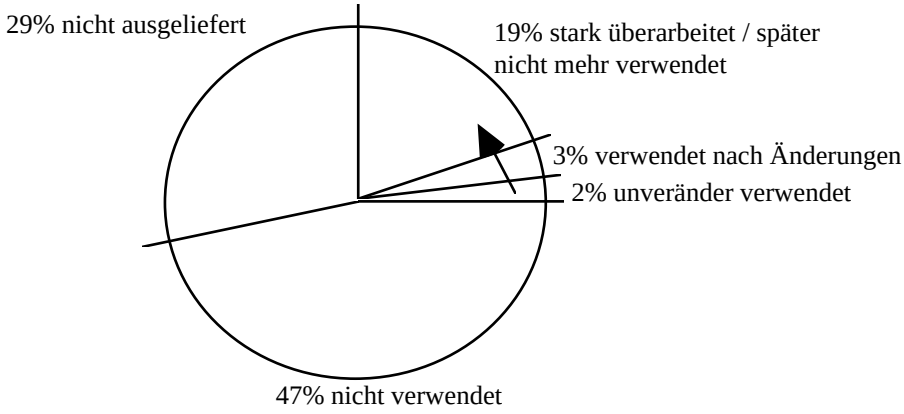
- Bedarf an umfangreicherer komplexer Software nimmt zu
  z.B. grafische Oberflächen, Datenbanken, Kommunikation, ...
- Projekte werden komplexer und unüberschaubarer
  z.B. Automatisierungsprojekte, Unternehmensdatenmodelle, ...
- Lebensdauer von vielen Softwareprodukten ist relativ gering
  z.B. Versionen von DOS, Turbo-Pascal, Word, ... *außer bei IBM*
- Hardware ist sehr viel billiger geworden, aber auch kurzlebiger:
  Abschreibung auf Bürocomputer 3 Jahre
- Gemeinkosten und Lohnkosten (Lohnnebenkosten) sind stark gestiegen
  d.h. Miete, Kapitalkosten, Löhne, ...



Problem: trotz hoher Erstellungskosten haben die Systeme Mängel:

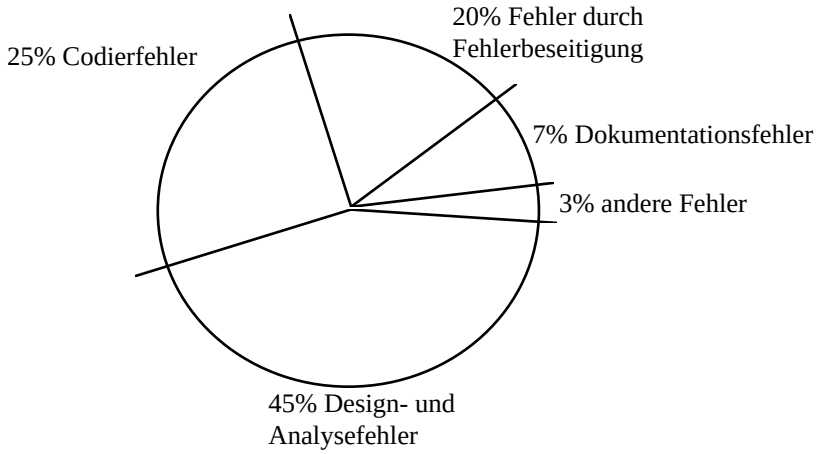
- sie sind meist fehlerhaft,
- erfüllen / treffen nicht die Anforderungen der Benutzer,
- werden zu spät fertiggestellt,
- werden meist teurer als geplant.

Viele Systeme können erst nach umfangreichen Änderungen eingesetzt werden *wenn überhaupt, viele enden in der Schublade oder in der "Rundablage unterm Tisch".*


29% nicht ausgeliefert
19% stark überarbeitet / später nicht mehr verwendet
3% verwendet nach Änderungen
2% unveränder verwendet
47% nicht verwendet

Quelle Comp.Wo. 15.4.88

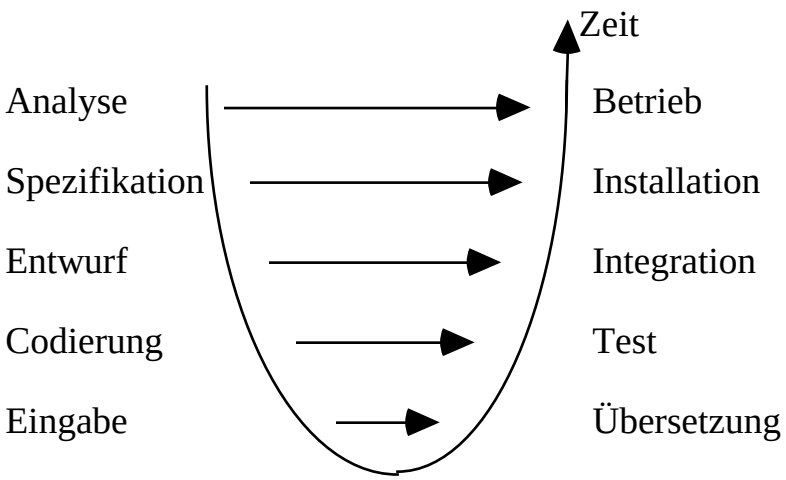
Die meisten Fehler entstehen **nicht** während der Programmierung, sondern lassen sich auf Fehler in früheren Phasen zurückführen, *wenn es diese Phasen überhaupt gegeben hat.*



Häufig pflanzen sich Fehler in Folgephasen fort, *nach Murphy in sonderbarer, fast mysteriöser Weise!*

Je später Fehler entdeckt werden, desto höher sind die Kosten für ihre Beseitigung.

Fehler können nur auf derjenigen Abstraktionsstufe gefunden werden, auf der sie gemacht wurden:



Dies führt dazu, daß Fehler aus frühen Phasen relativ lange unbemerkt bleiben, bevor sie entdeckt werden.

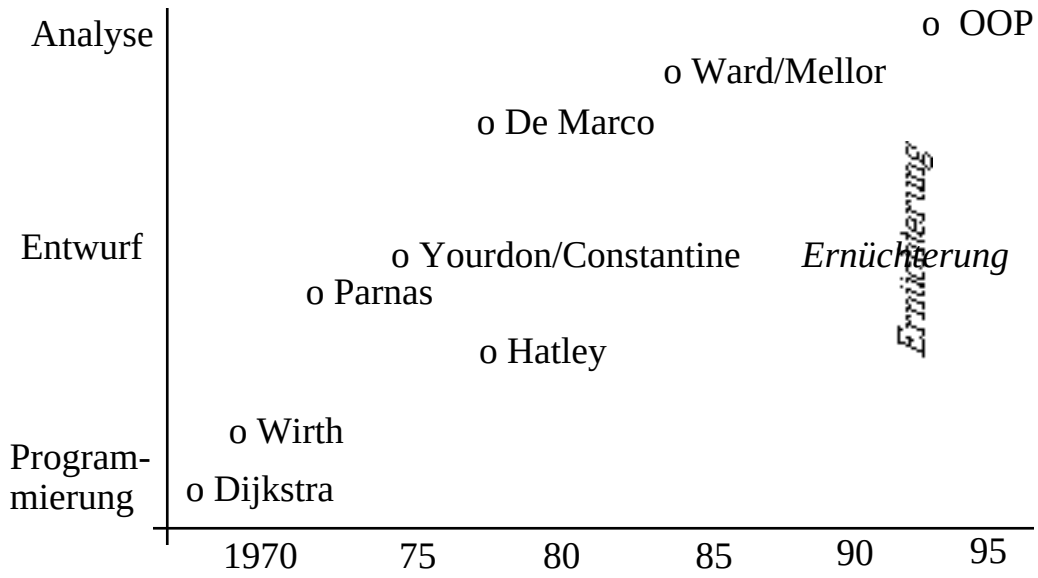
Bereits 1968 und 1969 fanden zwei NATO-Konferenzen statt, auf denen Lösungsansätze entwickelt wurden, um die **Softwarekrise** zu lösen.

Zur Lösung der aufgezeigten Probleme begann man, strukturierte Methoden zu entwickeln. Erste Ansätze befassten sich ausschließlich mit der Codierung.

Dijkstra: *hört endlich auf mit GOTO zu programmieren!*

Die in der Folgezeit entwickelten Methoden und Techniken befassen sich mit der "ingenieurmäßigen" Entwicklung von Software.

\+ Sie werden unter dem Oberbegriff **Software-Engineering** (bzw. **Programmkonstruktion**) zusammengefaßt.



Dijkstra:	Programmverifikation, sequentielle/parallele Prozesse, kein goto mehr
Wirth:	Strukturierte Programmierung (PASCAL, Modula-2)
Hatley/Pirbhai:	Real-Time Modelling
Parnas:	Einführung des Begriffs "Modul" und Definition
Yourdon/Constantin:	Structured Design Technique (SD)
De Marco:	Structured Analysis (SA)
Ward/Mellor:	Erweitererung der SA (De Marco) um Kontrollprozesse
Chen:	Entity Relationship Approach
XEROX Corp:	Smalltalk ab 1970 entworfen, 1983 Smalltalk-80 von A. Goldberg + D. Robson
Stroustrup:	OOP mit C++, 1983 bei AT&T
Booch/Jakobson/ Rumbaugh	OO-Analyse und -Design mit UML, 1995

<u>Prinzipielles Problem:</u>

Je komplexer die Systeme werden, desto:

- schwieriger wird die Überwachung der Qualität der SW und um so
- unüberschaubarer sind die Auswirkungen nachträglicher Änderungen.

Die Programme werden stets komplexer, größere Rechner lassen dies zu (jedes Jahr um den Faktor 2, wie bei Speicherchips und Mips?), der Mensch kann in seiner bescheidenen intellektuellen Leistungsfähigkeit dabei nicht mithalten.

Das führt zu dem Engpaß:

oft versteht nur noch der Programmierer sein Programm,
andere natürlich auch, aber erst nach Monaten des Grübelns.

Selbst der Programmierer versteht sein Programm nicht mehr, wenn er sich längere Zeit nicht damit beschäftigt. Damit wiederum werden die Kosten für Systemwartung zunehmend größer und damit auch wichtiger. Hinzu hinzu kommt noch:

- **Wartung ist oft Beseitigung von verschleppten Fehlern!**

Ziel jeder industriellen Software-Erstellung ist (sollte sein):

das Erreichen einer	**• hohen Produktivität und**
	• Qualität
unter Einhaltung der	**• geplanten Kosten und**
	• Termine.

Anmerkung: In Forschung (und Lehre) findet in der Regel keine industrielle Software-Erstellung statt, weil das Ziel oft offen ist und der Einsatz "keine" Rolle spielt.

Das Ziel kann nur erreicht werden, wenn die verschiedenen Aufgaben von:

- Entwicklung	(Sprachen u. Werkzeuge, Tools, ...
- Qualitätssicherung	(tatsächlicher Stand der SW, Zeit, Kosten,...
- Management	(Organisationsmodelle, Kommunikation,...

den Projektteilnehmern bewußt sind.

Na, wie weit sind sie denn, Herr Müller?

I) fast fertig, bis auf ... d.h. muß noch mind. 30% machen

II) habe gerade erst angefangen und... d.h. hat ca. 60% fertig

Für viele Methoden der Softwareerstellung und Projektabwicklung insgesamt gibt es Werkzeuge, sogenannte **CASE-Tools**, (CASE = Computer Aided Software Engineering):

- Software Through Pictures (STP)
- Innovator
- Rational Rose

Jedoch:

Werkzeuge helfen nur denen, die sie richtig anwenden *können (wollen).*

Wichtiger noch als die Werkzeuge selbst ist das Wissen um:

- die Prinzipien (welche zu beachten sind)
- die Methoden (welche richtig einzusetzen sind)
- die Werkzeuge (welche es gibt, wann sinnvoll einsetzbar)
- Organisationsmodelle (welche Struktur für welche Aufgabe bzw. Projektphase)

Zur bewußten Unterscheidung:

Prinzipien:
sind Grundsätze, die wir dem Handeln zugrunde legen.
Sie sind allgemeingültig und abstrakt.
Sie entstehen aus Erfahrungen und Erkenntnissen.

z.B. schlechte Prinzipien:
- Sage deinem Chef nie, wie weit du wirklich bist.
- Mache dich durch trickreiche Programmierung unentbehrlich.

Zu den guten Prinzipien kommen wir noch.

Methoden:
sind planmäßig angewendete, begründete Vorgehensweisen zur Erreichung von festgelegten Zielen.
Sie enthalten also den Weg (d.h. ein Rezept) zu etwas hin.
Sie machen Prinzipien anwendbar.

z.B. schlechte Methoden:
- Man versorge seine Kollegen stets reichlich mit Spielprogrammen.

z.B. gute Methoden:
- Man zeichne einen Programm-Ablaufplan (PAP).

Werkzeuge:
bzw. Hilfsmittel unterstützen die eingesetzten Methoden oder Verfahren.
Sie sollen den Einsatz von Methoden oder Verfahren
erleichtern, beschleunigen, sicherer machen, ...

z.B.: make, RCS, ..., allgemein: CASE-Utilities.

Organisationsmodelle
ToDo

Allgemein:

Der Entwurf von Software ist ein kreativer Akt !

Es gibt keine Patentrezepte, *Gott sei Dank.*

Vergleich mit dem Bau eines Hauses:

Es gibt Architekten und Handwerker, jeder ist Spezialist auf seinem Gebiet:

⇐ Rohre schweißen, Elektroinstallation, usw. *besser nicht von Architekt!*

⇐ Gesamtplan erstellen (Entwurf) und überwachen (Projekt-Management)
ab 4-stöckigem Haus besser nicht von Handwerker!

Ein *guter* Architekt kann ein 4-stöckiges Haus mit 20 Mann auch ohne Plan bauen.
Besser ist es jedoch, erst einen Plan zu erstellen und dann zu beginnen.

Methoden: Bauplan und Netzplan
Werkzeuge: CAD für Statik, Netzplanprogramm für Ressourcenverwaltung und Zeitüberwachung

Aber: die geistige Arbeit wie

Analyse	*d.h. wo der Eingang, Treppe, Fenster ...*
Entwurf	*von welcher Firma die Glaselemente ...*

muß er letztendlich selbst machen, *oder auf alte Pläne zurückgreifen (reusability).*

2 Programmentwicklungszyklus

Alle während der SW-Entwicklung und anschließender Wartung und Pflege anfallenden Einzelaktivitäten werden zu Tätigkeitsgruppen zusammengefaßt. Jede Gruppe produziert ein oder mehrere Teilprodukte des Gesamtsystems.

⇐ *Es gibt Schnittstellen zwischen vorausgehenden und folgenden Produkten.*

Tätigkeitsgruppen und ihre Produkte werden als **Phasen** zusammengefaßt. In der Praxis existiert eine Vielzahl von Phasenkonzepten, welche sich durch

- die Phasenbezeichnungen,
- die enthaltenen Tätigkeiten und
- den Detaillierungsgrad

unterscheiden.

Wir gehen von nachfolgender Phaseneinteilung aus:

1 Problemanalyse und -spezifikation

Vor der eigentlichen Entwicklungsphase des SW-Produkts wird das Produkt grob vorgeplant und kalkuliert.
Ergebnis kann sein: überhaupt nicht machen, besser einkaufen.
Danach werden die genauen Anforderungen an das Produkt spezifiziert *zusammen mit Anwender.*

3 Entwurf

Hier wird eine softwaretechnische Lösung im Sinne einer Systemarchitektur entwickelt.

4 Implementierung

Die einzelnen Teile werden in einer (oder verschiedenen) zweckmäßigen Programmiersprache(n) geschrieben.

5 Funktions- und Leistungsüberprüfung

Nach der Integration aller Komponenten erfolgt der Gesamttest
evtl. Tuning, Redesign, ...

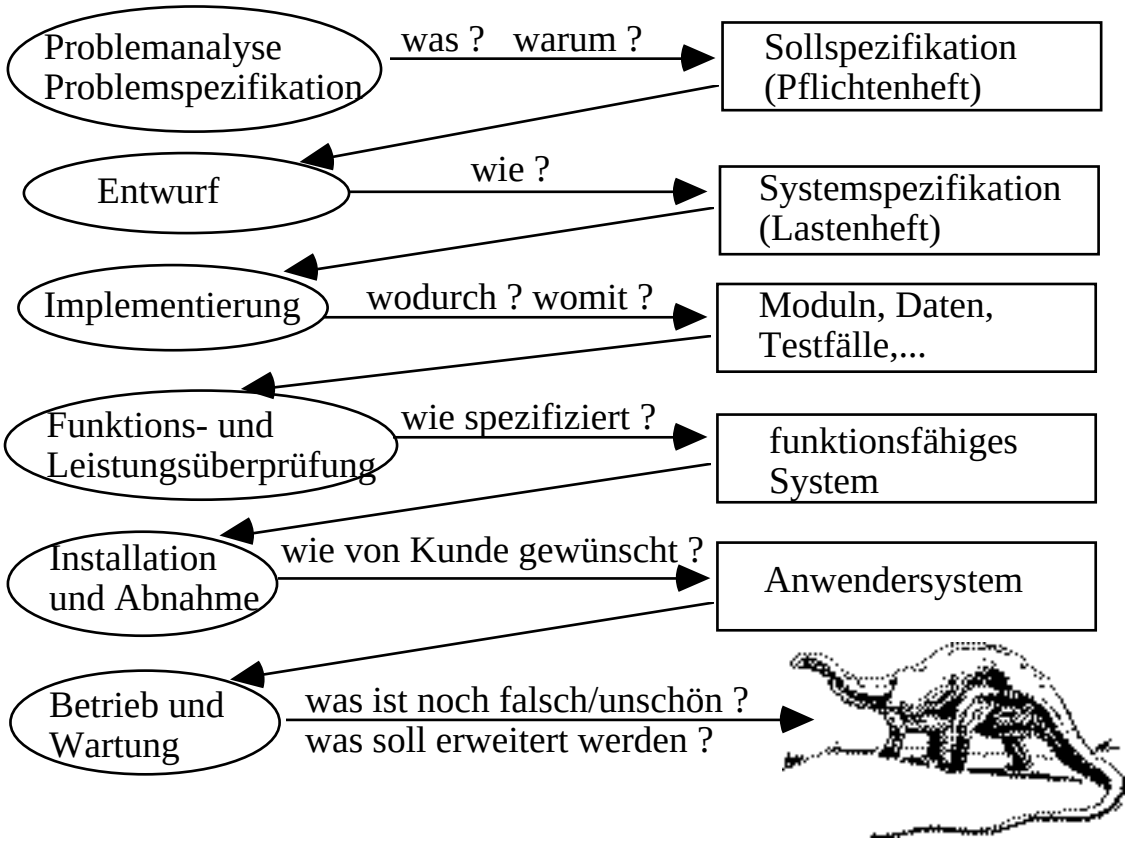
6 Installation und Abnahme

Das Produkt wird beim Anwender installiert und nach einer Einführungsphase erfolgt die Abnahme.

7 Betrieb und Wartung

Während des Betriebs auftretende Fehler werden beseitigt (Pflege), möglicherweise muss das Produkt aber auch angepaßt und geändert werden (Wartung).

Die einzelnen Phasen bauen wie folgt aufeinander auf:



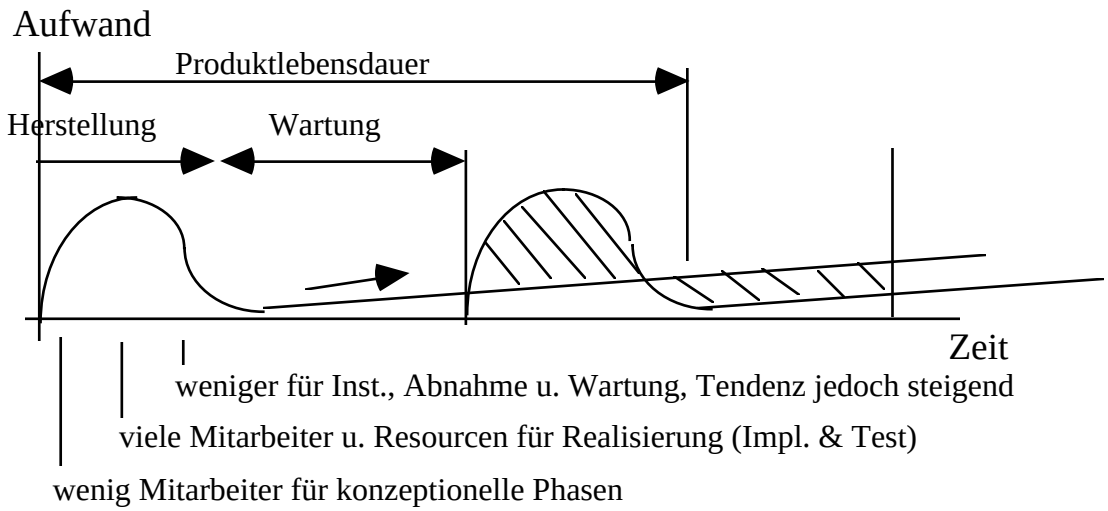
Wenn man dieses Bild etwas anders zeichnet, dann wird klar, warum man oft auch vom **Wasserfall-Modell** der Software-Entwicklung spricht.

Jede der Phasen kann nochmals in je 3 Einzelschritte aufgeteilt werden:

- **Planung** verfeinert den Rahmenplan
- **Durchführung** produziert alle Teilprodukte
- **Überprüfung** sichert die Qualität aller Teilprodukte

Treten innerhalb der Phasen Betrieb und Wartung über die Zeit mehr und mehr Änderungswünsche der/des Kunden auf, so wird es in der Regel immer aufwendiger, diese **nachträglichen**, **zusätzlichen** Leistungen in das bestehende Programmsystem zu implementieren.

Es gibt einen Punkt, ab dem eine Neuentwicklung **finanziell** günstiger ist: wenn die beiden Flächen im Diagramm gleich groß sind (sie entsprechen gerade den Kosten).



2.1 Problemanalyse und -spezifikation

Diese Phase besteht aus den Teilen: Istanalyse, Sollkonzept, Durchführbarkeitsstudie, und Planung.

Istanalyse

- Erfassung des momentanen Zustands
- Abgrenzung des betrachteten Problems vom Rest
- Definition der Umgebung soweit erforderlich (Schnittstellen, Randbedingungen)

Sollkonzept

- wichtigste Teilphase: die Spezifikation des Problems, bzw. der Aufgabe
- basierend auf dem momentanen Zustand muß der künftig gewünschte Zustand zusammen mit dem Anwender besprochen und festgelegt werden
- ggf. mehrere Alternativen entwickeln

Durchführbarkeitsstudie

- Lösungsansätze müssen ausgearbeitet und auf ihre Durchführbarkeit hin untersucht werden
  - entsprechende Umgebung / Ausstattung vorhanden
  - notwendiger Aufwand / Kosten
- Entscheidung, ob das Projekt gestartet wird oder nicht

Projektplanung

- Es werden alle Phasen des weiteren Projekts in ihrem zeitlichen Ablauf detailliert geplant.
- Der Bedarf an Betriebsmittel, Personal, Geräte, ... für die folgenden Phasen wird berechnet.

2.2 Entwurf

In der Analysephase werden die Anforderungen so beschrieben, daß sie bewußt keine Hinweise auf eine konkrete Realisierung enthalten. Dies begünstigt die Übertragbarkeit der Lösung auf beliebige Rechnersysteme.

Innerhalb der Entwurfsphase entwickelt man ein Modell des gesamten (Software-) Systems, welches umgesetzt in ein Programm, die gestellten Anforderungen im Pflichtenheft erfüllt.

Hierbei wird das komplexe Gesamtsystem in

- unabhängig voneinander realisierbare,
- in ihrer Komplexität überschaubare,
- annähernd gleich große Einzelbausteine (Module)

zerlegt. Man spricht daher auch von Modularisierung.

Die Vorteile der Modularisierung sind:

- Parallelisierung des weiteren Entwurfs
- Parallelisierung der folgenden Implementierung
- Wiederverwendung bereits vorhandener Bausteine (soweit möglich)
- leichte Austauschbarkeit von einzelnen Modulen
- Beschränkung des Wissens über andere Module bei der Implementierung

Die Modularisierung soll also:

- die Problemlösung verständlich und ihre Korrektheit nachprüfbar machen
- die unabhängige Implementierung von Modulen sicher stellen
- die Austauschbarkeit von Modulen mit gleicher Funktion ermöglichen

Die klassischen Verfahren für den Entwurf sind das

- **Structured Design** bzw.
- **Modular Design.**

Zunehmend an Bedeutung gewinnt das Verfahren

- **Object-Oriented-Design (OOD),**

das allerdings in anderen Vorlesungen ausgeführt wird.

2.3 Implementierung

> Wurde der Entwurf richtig durchgeführt, so wird ein hoher Grad an Parallelität bei der Implementierung erreicht.

Ergeben sich keine gravierenden Fehler bei der Definition der Schnittstellen, so kann jeder Programmierer relativ unabhängig die ihm übertragene Implementierungsarbeit ausführen. Die projektinterne Kommunikation ist ab dann auf das Notwendigste begrenzt, Absprachen untereinander (welche in der Praxis nur noch selten dokumentiert werden) entfallen weitgehend.

Unter der Impementierung wird die

Realisierung des Entwurfs mit Hilfe einer Programmiersprache verstanden.

Im allgemeinen ist der Programmierer frei in der Wahl einer Programmiersprache, sofern der Anwender nur am ausführbaren Programm interessiert ist und keine Rechte am Quellcode für sich beansprucht. Wenn der Anwender eine spezielle **Implementierungssprache**, ein **Betriebssystem** oder eine **Zielhardware** festlegt, so ist dies stets Bestandteil des Pflichtenhefts.

Ab dieser Phase kann sich der Einfluß einer gewählten oder vorgeschriebenen Zielhardware bzw. Quellsprache auf die Qualität des Programms auswirken. Beispielsweise gibt es nicht für alle Rechner Übersetzer für jede Sprache. Treten Effizienzkriterien hinzu, so scheiden u.U. weitere Sprachen aus. Gibt es diesbezüglich keine Einschränkungen, so sollte eine möglichst problemnahe Sprache verwendet werden, welche eine effiziente Implementierung sicher stellt.

2.4 Überprüfung und Test

Beim Testen wird ein (Unter-)Programm mit Testdaten "gefüttert" (Stimuli) und seine Ausgaben werden mit den erwarteten Ausgabedaten (Response) verglichen. Tritt eine Abweichung ein, so liegt ein Fehler vor.

Das Grundtheorem des Testens von Dijkstra lautet jedoch:

> Program testing can be used to show the presence of bugs,
> but never their absence.

Darum stellt das Testen nur einen schwächeren Ersatz für einen formalen Korrektheitsbeweis (Verifikation) eines Programms dar.

Es gibt praktisch keine größeren Programme, welche vollständig fehlerfrei arbeiten, obwohl die meisten davon durchaus mehrschichtig und gründlich getestet wurden.

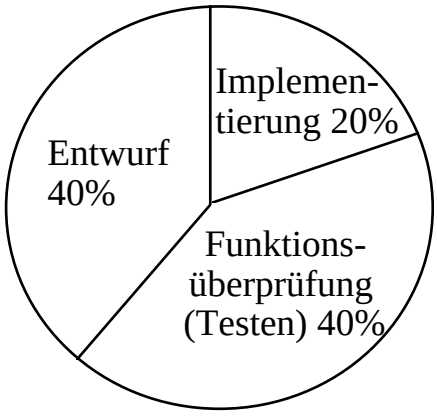
Entsprechend der hierarchischen Gliederung des Entwufs in einzelne Module, welche später zur Lösung zusammengefügt werden, unterscheidet man beim Funktionstests den

- **Modultest** und
- **Integrationstest,**

sowie den sich anschließenden

- **Leistungstest.**

Der Aufwand, welcher zur Vorbereitung und Durchführung von Tests benötigt wird, darf nicht unterschätzt werden:



Quelle Inform. Duden

Insbesondere ist die Ermittlung von aussagekräftigen Testdaten keineswegs einfach und sollte bereits in der Phase des Entwurfs erfolgen.

2.5 Installation und Abnahme

Bei der Installation wird das Softwareprodukt in seine

Zielumgebung verpflanzt

und übernimmt dort seine Aufgabe. Dazu sind i.d.R. organisatorische Umstellungen, Konvertierung und/oder erstmalige Erfassung von Datenbeständen, Installation von Infrastruktur, ... notwendig.

Gelegentlich muß Motivationsarbeit geleistet werden, um die Mitarbeiter von dem neuen System zu überzeugen.

Sind die Benutzer keine DV-Fachleute, so ist oftmals eine Schulung für die

- normalen Benutzer und den
- späteren Administrator des Systems *(local Guru)*

notwendig.

Die **Installation stellt primär ein innerbetriebliches, organisatorisches Problem dar**, Support und Entwicklung können jedoch ihren Beitrag zum reibungslosen Ablauf und zur Annahme des Systems leisten.

Nach erfolgter Installation, Schulung und Aufnahme des Betriebs erfolgt die Abnahme des Systems durch den Auftraggeber, bei innerbetrieblichen Entwicklungen durch die entsprechende Abteilung oder die Verkaufsabteilung und den technischen Support.

Bei größeren Auftragsprojekten sollte die Abnahmeprozedur vor dem Vertragsabschluß festgelegt werden und somit Bestandteil des Pflichtenhefts werden. *Man erspart sich damit manchen Ärger!*

Der Auftraggeber überprüft, ob die im Pflichtenheft aufgeführten Funktionen und Leistungsmerkmale vorhanden sind und das System auch (hoffentlich) seinen Vorstellungen entspricht.

Während oder nach der Abnahme erfolgt häufig noch eine **Feinanpassung** (**Tuning**) des Systems an die speziellen lokalen Gegebenheiten:

- Einstellung von Netzwerkadressen, Backupzeiten, ...
- lokale Peripherie anschließen und in Betrieb nehmen
- Passwörter einstellen, ...
- in der PDV oft umfangreichere Einstellung der Anlage, wie etwa Verzögerungszeiten von mechanischen Teilen, Empfindlichkeit von Sensoren, etc.

2.6 Betrieb und Wartung

Im Betrieb zeigen sich die eigentlichen Qualitäten eines Produktes wie

Funktionsfähigkeit, Ausfallsicherheit und Leistung.

Bei der fast immer notwendigen Wartung zeigen sich die Qualitäten wie

guter Entwurf und Modularität.

Die Wartung stellt zwar das letzte Glied im Software-Live-Cycle dar, jedoch entfällt auf dieses anteilig am Gesamtprojekt ein hoher Anteil (ca. 40-70%) aufgrund

- der langen Laufzeit dieser Phase und
- des multiplikativen Faktors, wenn das Softwareprodukt mehrmals ausgeliefert wurde (was der Regelfall sein sollte).

Neben der Beseitigung von (bei größeren Systemen unvermeidlichen) Fehlern ist es die Aufgabe der Wartung, Anpassungen und Erweiterungen des Systems an die sich ändernden Anforderungen vorzunehmen.

Fehlerkorrekturen und mehrfache Erweiterungen und Änderungen über die Jahre führen zu einem zunehmend unübersichtlicheren System *(Saurier)*.

Durch die zunehmend überproportionale Bindung von Personal steigen die Kosten für jede weitere Änderung stark an, so daß schließlich ab einem gewissen Alter der Software eine Neuentwicklung langfistig kostengünstiger ist als eine Ausdehnung der Wartung.

Man spricht hier gelegentlich von einem Verschleiß der Software.

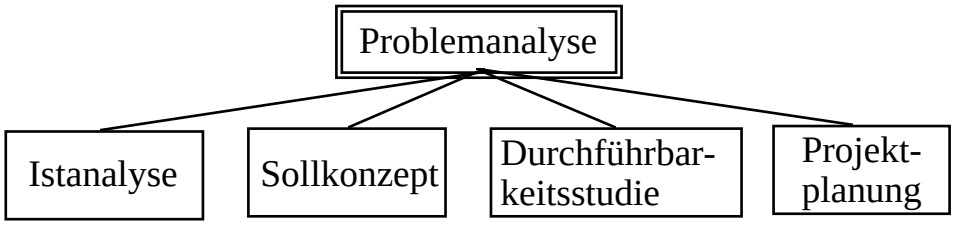
Der Software-Live-Cycle schließt sich mit der Neuentwicklung eines Systems, welches das alte ablösen soll.

3 Problemanalyse

Ziele der Analysephase:

- Das zu lösende Problem und alle wichtigen Umgebungsbedingungen müssen vollständig und eindeutig erfaßt werden.
- Die Durchführbarkeit und Wirtschaftlichkeit der geplanten Softwareentwicklung müssen untersucht werden.

Die Problemanalyse gliedert sich in vier Teile:



3.1 Istanalyse

Die Istanalyse soll die bestehende Systemstruktur beschreiben.

Systemabgrenzung:

Was gehört nicht mehr zum System dazu?
Was sind wichtige Randbedingungen, die es zu beachten gilt?
Wann funktioniert das System nicht mehr?

Vorsicht, dabei werden die für das Problem relevanten Teilaspekte der Führungs- und Entscheidungsstruktur des Unternehmens aufgedeckt.

Systemerhebung

- Bereits existierende Informationsflüsse und -kanäle erfassen.
- Analyse der vorhandenen Informationsverarbeitungsverfahren:

Wie komplex ist eine (Teil-) Aufgabe?
Läßt sie sich schematisieren (mechanisieren)?
Wie weit ist sie bereits automatisiert?
Wodurch wird die Aufgabe angestoßen?
Welche Ein- und Ausgabedaten hat die Aufgabe?
Wie wird sie momentan gelöst?
Welchen Umfang haben die Daten?
Welche Verarbeitungsgeschwindigkeiten liegen vor?

Systembeschreibung

Nachdem alle Zusammenhänge und Abhängigkeiten erfaßt sind, kann ein erstes reduziertes Modell erstellt werden.

Faktenanalyse

Sie liefert genaues Zahlenmaterial, mit dem das Modell auf seine Richtigkeit und Vollständigkeit hin überprüft werden kann.
#Leseoperationen/sec, #Tansaktionen/sec, #User mittel/max

Das Hauptproblem bei der Istanalyse ist die Gewinnung von zuverlässigem Material, weil Betroffene oftmals

- Widerstand gegen die Einführung des Projektes leisten, bzw.
- pessimistische Einstellung gegenüber dem Projekt haben (Arbeitsplatz gefährdet, Selbstwertgefühl verletzt, ...)

Man muß daher mit unrichtigen, verzerrten Daten und Auskünfte rechnen, darf keine Unterstützung erwarten und muß oft sogar Behinderungen in Kauf nehmen.

Die Einstellung ist auch erklärbar, da das Know-How des Einzelnen übertragbar wird auf Maschinen:

- *neuronale Netze für Börse*
- *Experten-Systeme für Kapitalanlage/Steuer/Medizin/Recht/...*

3.2 Sollkonzept

Das Sollkonzept oder die Sollanalyse

- ist die wichtigste Phase innerhalb der Problemanalyse,
- ist auch bei kleinen übersichtlichen Projekten **unverzichtbar.**

Die Erstellung des Sollkonzepts umfaßt die Formulierung der **Systemziele**, d.h. der wesentlichen Aufgaben des Systems.

Ziele können vielfältig sein:

- *Kostenverminderung (auch Personaleinsparung)*
- *Geschwindigkeitssteigerung*
- *Erhöhung der Sicherheit*
- *Erhöhung der Produktqualität*
- •••

Überlegungen bei der Definition der Ziele:

1) Benutzermodell

Anforderungen an den künftigen Benutzer des Systems (Anwender)

- Qualifikation, Schulungen, auch Rechnerkenntnisse,...

2) Basismaschine

Minimalforderungen an die später eingesetzte Rechenanlage:

- Systemsoftware
- zus. Betriebsmittel (Plattenplatz, Band, Netzwerk,...
- zus. Peripherie (Farb-, Bildschirm, Maus, Tablett, Barcode,...
- Verarbeitungsgeschwindigkeit (Mips, Transaktionen,...
- Ausfallsicherheit (Notstrom,...

3) Benutzerschnittstelle

Zugang eines Benutzers, Identifikation, Datenschutz,...

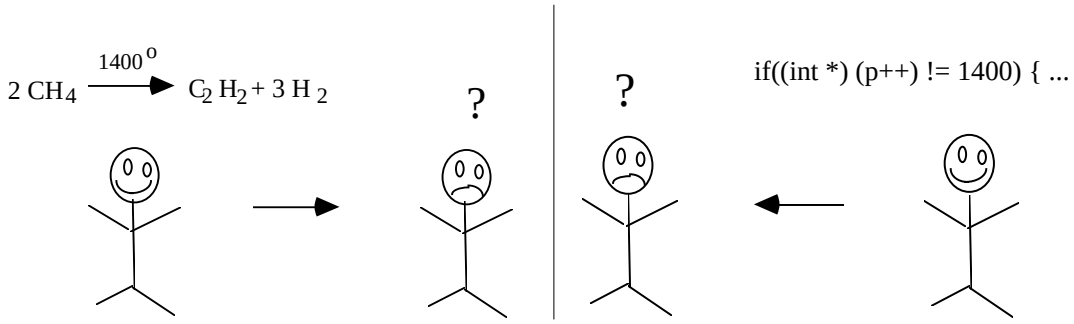
Bedienung des Systems, aufgeteilt auf die verschiedenen Benutzerklassen

z.B. Warenhaus: -Verkäufer
- Storno für Abt.Leiter
- Bestellungen
- Zahlungsanweisungen

Verhalten bei auftretenden Störungen

Notmaßnahmen

Hauptproblem bei der Analysephase: jeder hat zunächst nur seine Sicht auf das Problem.



Oftmals werden mehrere alternative Lösungskonzepte erarbeitet, jedoch **stets unter Einbeziehung des Anwenders.**

. MERKE:

1) Nur selten wird ein Problem zum ersten Mal gelöst,...

2) Auch wenn jedes Problem individuell gelöst werden muß, so gibt es doch i.d.R. eine Vielzahl ähnlicher, bereits gelöster (Teil-) Probleme.

3) Oftmals kann es sinnvoller sein, eine Standardlösung (fertiges Produkt) einzukaufen oder die Lösung in Zusammenarbeit mit einem anderen Unternehmen zu erstellen.
Spezialisierung bzgl. Hardware (SUN, HP, NeXt,...), Software (PDV, DTP,...)

4) Oftmals ist der Mißerfolg des Einsatzes von EDV darauf zurückzuführen, daß man versucht, die aus der Handarbeit gewohnte Organisation, Arbeitsabläufe, technische Verfahren in die Datenverarbeitungsphase **hinüberzuretten.** Das betrifft äußere Verfahren, wie auch rechnerinterne Abläufe.

5) Oftmals existieren in der Übergangsphase alte und neue Lösung, dies bedeutet zusätzliche Arbeit (auch Kosten), weil zwei Systeme parallel betrieben werden müssen, darum: Dauer der Übergangsphase planen!

+ Fazit:

> *Erfolg oder Mißerfolg eines SW-Projektes ist oft abhängig von gründlicher Vorplanung.*

3.3 Durchführbarkeitsstudie

Für die erarbeiteten Ziele und Lösungsvorschläge muß geprüft werden, ob:

- sie überhaupt technisch realisierbar sind
  - ***** hinreichend großer, schneller, sicherer, ... Rechner
  - ***** Möglichkeit der Vernetzung
  - ***** Umgebungsbedingungen: Temperatur, Staub, EMV,...

- die notwendigen Randbedingungen geschaffen werden können
  - ***** technischer Natur
  - ***** personelle Voraussetzungen
  - ***** soziales Umfeld *(z.B. Personalrat bei ISDN)*

- sie ökonomisch vertretbar sind, d.h innerhalb eines vorgegebenen Rahmens ausgeführt werden können
  - ***** Terminvorgaben
    - *- Urlaubszeit in Automobilbranche,*
    - *- zus. Mitarbeiter aus anderer Abteilung für Übergangszeit*
    - *- Mitarbeiterschulung noch vor Weihnachtsgeschäft*
  - ***** Kostenvorgaben
    - *- Mittel für Beschaffung Rechner in 7/93*
    - *- zus. Personal binden für n Monate ab 8/93*

Das Ergebnis der Durchführbarkeitsstudie führt

- zur **Aufgabe** des Projekts
- zur **Revision der Anforderungen** oder

- zur **Durchführung** des Projektes.

3.4 Projektplanung

Zum Abschluß der Problemanalyse erfolgt die Projektplanung. Sie umfaßt die Formulierung von Grob- und Teilzielen, welche innerhalb eines zeitlichen Rasters angesiedelt werden. Projekt- und alle Teilziele sind fest, sie können sich grundsätzlich nicht aus der Arbeit am Projekt ergeben.

Aus dem Projekt heraus kann jedoch der Anstoß zur getrennten Analyse von
- Ideen
- neuen Produktvorschlägen (oder -verbesserungen) sowie
- effizienteren Lösungen

gegeben werden.

Dies sind dann jedoch neue Projekte und haben mit der aktuellen Aufgabe nichts gemein. Sie können evtl. im Rahmen von Vorlaufforschung/Entwicklung als interne Projekte definiert werden.

In der Planung muß genau zwischen einem Forschungs- und einem Entwicklungs- oder Kundenprojekt unterschieden werden.

Bei einem Entwicklungsprojekt muß die Planung garantieren, daß das Entwicklungsziel auf jeden Fall erreicht wird.

Hierzu müssen bereits Methoden zur Erfüllung aller Teilziele verhanden sein. Bei Forschungsprojekten sollen oftmals gerade diese Methoden erst gefunden bzw. ausgearbeitet werden. Naturgemäß kann man hier nicht garantieren, daß das Ziel auch tatsächlich mit den gegebenen Mitteln (Zeit, Aufwand,...) erreicht wird.

Bei der Planung eines Entwicklungsprojektes werden alle benötigten Ressourcen quantitativ und zeitlich möglichst genau eingeplant:

Betriebsmittel:
- Netzwerkkomponenten
- Rechner, Rechenzeit, Speicherplatz,
- Spezialperipherie wie Plotter, Scanner, Fotobelichter, ...
- Entwicklungswerkzeuge wie Emulatoren, Analysatoren, ...
- die Beschaffung und Lieferung von Teilen
  - Bauelemente, Spezialkomponenten,...
  - Sensoren, Aktoren,... in der PDV
- Software-Produkte, Lizenzen für Compiler, Tools, ...
- Beschreibungen und Manuals
- Erwerb von evtl. (noch) nicht vorhandenen Wissen
- Einplanung finanzieller Mittel
- •••

Mitarbeiter (Kapazitätsplanung):

- Informatiker für Grob-, Feinentwurf
- Programmierer für einfache Codierungsarbeit
- Spezialisten für unterschiedliche Bereiche
  - Betriebssystemangelegenheiten
  - DTP, Oberflächen
  - Datenbankanwendungen
  - PDV-Bereich
  - Kommunikation
  - •••
- Spezialisten aus den Anwendungsbereichen

  z.B. Chemie, Masch.-Bau, Fertigungstechnik,...
- •••

Auch auf der Seite des Anwenders müssen ggf. Maßnahmen eingeplant werden:

- Infrastrukturmaßnahmen
- interne Vorbereitungen für eine Umstellung von Abläufen
- Schulungsmaßnahmen.

In der Regel werden hierbei unübersichtlich viele Einzelaktivitäten von einem Team parallel ausgeführt. Die exakte Planung aller Aktivitäten, Termine, Meilensteine, die Personaldisposition, usw. erfolgen insbesondere unter Berücksichtigung des Faktors Zeit mit geeigneter Rechnerunterstützung. Hierbei werden die meisten Termine und insbesondere auch (zeit-)kritische Wege innerhalb von Projekten ermittelt bzw. erkannt.

\+ Netzplanprogramme mit Kapazitäts- und Ressourcen-Management beispielsweise MS-Project.

Genauso wichtig wie die Planung ist auch die Projektkontrolle durch die Überprüfung des tatsächlichen aktuellen Projektstandes (**Projektverfolgung**) und die Überprüfung der Qualität der erzeugten Produkte (**Qualitätskontrolle**). Erst eine möglichst exakte Projektverfolgung und ein damit mögliches frühzeitiges Einlenken bei Problemen sichert das Erreichen der spezifizierten Projektziele.

3.5 Pflichtenheft

Das **Ergebnis** der Problemanalyse und darin insbesondere der Sollanalyse ist die:

> Präzisierung der Zielsetzung in Form eines Katalogs von Anforderungen und Randbedingungen, dem **Pflichtenheft.**

Pflichtenheft =

Anforderungsdefinition

+ zusätzliche Randbedingungen

+ zusätzliche Vereinbarungen wie z.B.

- Termine
- Abnahmeprotokoll
- Haftung bei Fehlern
  Produkthaftungsgesetz, Funktionsgarantie mit Recht auf Nachbesserung
- Aufgaben für Auftragnehmer
  Infrastruktur bereitstellen, erstmalige Datenerfassung,...
- evtl. Wartungsmodalitäten

Hinweise dazu:

Technik:
VDI / VDE- Richtlinien: **VDI/VDE 3694**
Deutsche Norm: **DIN ISO 9000, Teil 3:** Qualitätsmanagement- und Qualitätssicherungsnormen (entspricht unverändert der int. Norm!)

Öffentliche Auftraggeber in Deutschland:
Das **V-Modell**, Planung und Durchführung von IT-Vorhaben in der Bundesverwaltung, Koordinierungs- und Beratungsstelle der Bundesregierung (KBSt Band 27/1), Bundesminister des Innern (Überbl. in iX 3/95 S.162 ff: SW-Erstellung mit V-Modell)

International:
int. Norm: ISO 9000-3 Quality management and quality assurance standards
spez. in USA: DoD **STD 2167** (Verteidigungsindustrie)
Frankreich: GAM T 17

Im folgenden ist ein Schema für ein solches Pflichtenheft zu finden.

S.1:

Pflichtenheft zum Projekt

Prozeß-
informationssystem

Mai 1993

Auftraggeber: FHTE-Esslingen
vertreten durch: Herrn Dr. Hard *<- Zeichnungsberechtigter !*
Unterschrift:

Auftragnehmer: Fa. Mega-Soft
vertreten durch: Herrn Dipl.-Ing. Hacker *<- Zeichnungsberechtigter !*
Unterschrift:

Hinweis: I Pf. heft ist juristische Grundlage für den Vertrag (bzw. ist der Vertrag)
II Pf. heft darf nur in **beiderseitigem** Einverständnis verändert werden.

S. 2:

Inhaltsverzeichnis:

S. 3:

1. Allgemeine Beschreibung der Aufgabe

Verbale Zusammenfassung / Kurzform (max. 1/2 - 1 A-4)

S. 4 ff:

2. Genaue Beschreibung der Aufgabe (ist der Hauptteil !)

- zu realisierende Teilaufgaben und Zusammenhang
- Umfang, Betriebsmodi, Fenster, Darstellung, Ein- / Ausgabe ...
- auch was nicht mehr dazu gehört
- Beschreibung der Aufgabe (nicht der Lösung) per:
  SA (SA/RT), IM, ...
  bzw. Objekt-Orient. Analyse
- evtl. Aufzählung der Funktionen / Betriebsarten / Eingabe-möglichkeiten und deren Wirkung aus Anwendersicht

2.1 Funktion1, Betriebsart1, Ein/Ausgabe1... genau erläutern

2.2 Funktion2, Betriebsart2, Ein/Ausgabe2... genau erläutern

•••

3. Voraussetzungen beim Kunden

3.1 Systemumgebung

BS (HW-, SW-Voraussetzungen, Leistung, Sicherheit,
Speicher, Platte, notwendige SW-Tools , ...)

GUI (Auflösung Bildschirm, Anzahl Farben,
vorhandene Bibliothek, Schnittstellen, Compiler, ...)

DB (SQL-Datenbank oder sonstige Schnittstellen, ...)

LAN (TCP/IP, Novell, Leistung, Belastung, ...)

evtl. bekannte Probleme mit anderer SW

3.2 Noch vom Kunden zu erbringende Leistungen

- Datenbestände, Hardware-Ausbau, Infrastruktur,
  Notstrom, hinreichend Lüftung/Kühlung, Anschlüsse, ...
- Adressen, Kennungen, Datenerfassung, ...
- Kommunikations-Einrichtungen, Postleitung, ISDN, Funk, ...
- Genehmigungen einholen (auch Personalrat)

4. Abnahmeprotokoll

- die Funktionen A,B,C,... werden zur Abnahme vorgeführt
- Art u. Umfang der Dokumentation
- erworbene Rechte (Source, Anz. d. Lizenzen, Weitervertrieb, ...)
- Art/Umfang der Haftung bei Fehlern
  (i.d.R. ausgeschlossen, falls keine grob fahrlässigen Fehler)
- Möglichkeit der Nachbesserung binnen Frist von ...
- Fall des Verzugs (Konventionalstrafe DM/Tag, Ersatz, ...)
- Gerichtsstand

5. Zeitplan (nur grob, insbesondere *kein detaillierter. Netzplan* mit Interna!)

- wann Projektbeginn,
- evtl. Definition von Meilensteinen
  (z.B. Übernahme von Stammdaten vom Kunden, ...)
- wann Berichte, (Teil-) Zahlungen,...
- Beginn Installation, Probebetrieb, Abnahmetermin,
  Mitarbeiter-Schulung, Garantie, ...
- Garantiedauer, Hot-Line,

6. Wartungsmodalitäten (evtl.)

nur falls bereits Bestandteil des Vertrags, sonst nur Offerte
Update-Service / Kosten
Kosten bei Änderungswünschen
Support des Produktes bis... *(evtl. besser nicht ... !)*

7. Rücktrittsrecht

- nach Unterzeichnung des Pflichtenhefts kostenfrei für beide Partner bis spätestens ...
- danach jeweils nach zeitlicher Dauer am Gesamtprojekt

Nicht Bestandteil des Pflichtenhefts sind:

- Netzplan mit Aktivitäten u. Personal- / Resourcen-Planung
- Entwurf des Programm-Systems mit Unterprogrammen (Structure-Charts o.ä)
- Firmeninterna (soweit nicht unvermeidbar)
  z.B. - eigene Ausstattung / Infrastruktur
  - interne Personal-, Urlaubsplanung (wann, wer, was macht)
  - interne Stundensätze / Gemeinkosten / Gewinn ...

3.6 Zusammenfassung

Zur Darstellung von Daten und Abläufen gibt es eine Vielzahl von Modellen und Beschreibungsverfahren, die für die unterschiedlichen Phasen geeignet sind, z.B: Jackson Diagramme, HIPO-Charts, Warnier/Orr-Diagramme, ...

Wir werden uns mit den folgenden Beschreibungsverfahren beschäftigen:

- Strukturierte Analyse (SA) ♦ Tom De Marco, Ward/Mellor
- Realzeit-Modellierung (SA/RT) ♦ Hatley/Pirbhai
- Informations-Modellierung (IM) ♦ Chen (ER-Modell)

per Data-Dictionary (DD)
Bestandteil von SA bzw. SA/RT

4 Strukturierte Analyse

Mit Hilfe der Strukturierten Analyse (Structured Analysis, SA) lassen sich logische Systemmodelle konstruieren. Der Begriff „System“ kommt aus dem Griechischen und bedeutet ein „geordnetes Ganzes“ oder „gegliederte Vereinigung von Teilen“. Bereits an dieser Definition läßt sich erkennen, dass darunter viel mehr zu verstehen ist als nur ein Software-System. Mit Strukturierter Analyse lassen sich zumindest all die Systeme modellieren, die ein deterministisches Verhalten zeigen.

Modelle liefern keine exakte Abbilder der Wirklichkeit, sondern eine vereinfachte Darstellung der Funktion eines Gegenstandes oder des Ablaufs eines Prozesses, die eine Untersuchung oder Erfiorschung erleichtert oder erst möglich macht. Dennoch werden Modelle aus den genannten Gründen in der gesamten technischen Welt eingesetzt, um die wesentlichen Eigenschaften von Systemen herauszustellen. Sie werden gern benutzt, weil sie schnell und kostengünstig erstellt werden können und weil sie leichter zu ändern sind als die Systeme selbst oder Prototypen.

Die wesentlichen Eigenschaften von Software-Systemen sind die logischen Anforderungen (Essenz). Sie beschreiben die Eigenschaften, die das System haben muss, unabhängig davon, wie es später implementiert wird. Die Essenz eines Systems enthält alle technologischen Anforderungen, aber keine, die zur Erreichung der Aufgabe des Systems nicht benötigt werden. Logische Systemmodelle enthalten genau diese Anforderungen.

Häufig ergeben sich Schwierigkeiten beim Verfassen und Verstehen dieser rein textuellen und schlecht strukturierten Analysedokumente. Durch den systematischen Einsatz von Grafiken liefert das mit den Mitteln der Strukturierten Analyse angefertigte Systemmodell eine geeignete Grundlage, um Probleme, Handlungsalternativen und Entscheidungen zu diskutieren.

Structured Analysis (SA) gehört zu den Basistechniken zur Analyse und Spezifikation (im Großen). Sie ermöglicht es, die Gedanken bzgl. eines zu entwickenden Systems möglichst allgemein

- zu sammeln,
- zu strukturieren und
- schriftlich zu fixieren.

Die verwendete Sprache muß dazu sehr intuitiv, d.h.

- einfach zu verstehen,
- leicht zu erlernen und
- unkompliziert, übersichtlich und schnell in der Notation sein.

Dazu gibt es eine grafische Sprache mit wenigen Sprachelementen, mit denen Datenflußdiagramme (Data Flow Diagram, **DFD**) gezeichnet werden, sowie eine Notation für Prozeßspezifikationen (**P-Spec**) und für Datenstrukturen (Data Dictionary, **DD**). Datenflußdiagramme lassen sich miteinander verknüpfen, um die Systemhierarchie darzustellen.

4.1 Beispiel eines Systemmodells

In der (Problem-) Analysephase soll
- ein System als Teil eines Ganzen abgegrenzt, und
- die Beziehungen zum umgebenden System erfaßt und analysiert werden.

Dies wird an dem folgenden Beispiel demonstriert.

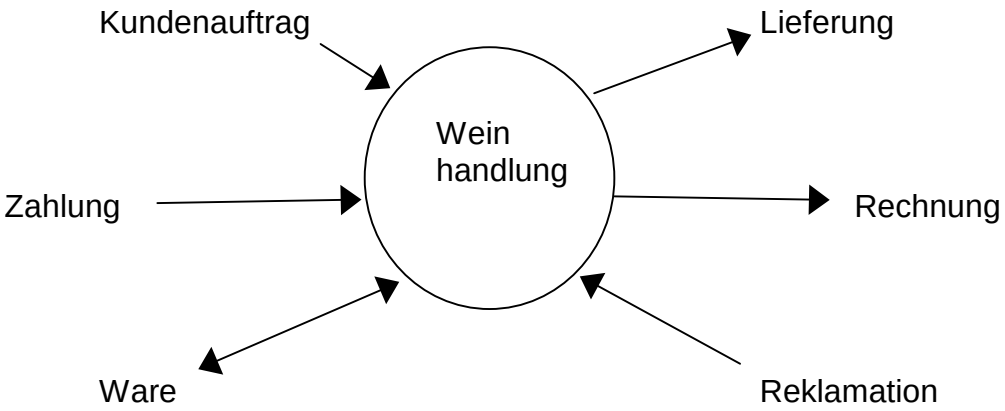
Beispiel: Weinhandlung

Umgebung: Lieferungen, Kundenaufträge, Zahlungen, Rechnungen, ...

Intern: Lager, Buchhaltung, Bestellannahme, Einkauf, ...

Aufgabe: Automatisierung von Kundenverwaltung, Lagerbestand, Ein- u. Verkauf, ...

Eine genaue Analyse umfaßt die Beschreibung der Schnittstellen nach außen, den sogenannten **Kontext**:



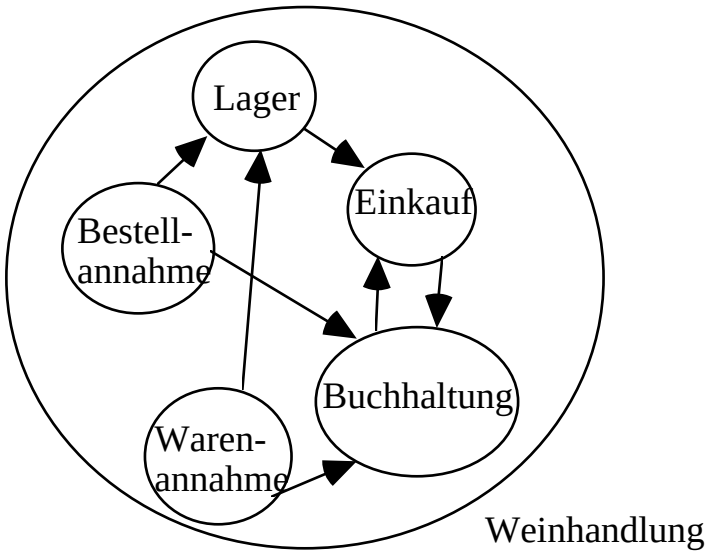
Weiterhin muss geklärt werden, wohin Lieferungen gehen, woher Reklamationen kommen, etc. Das sind die sogenannten Quellen und Senken eines Systems, die Terminatoren. Am Beispiel Ware ist dies zu sehen: Es kommt Ware von einem Lieferanten (Winzer etc.), Ware geht aber auch weg und zwar zu Kunden.

Im weiteren Verlauf ergeben sich etwa noch folgende Fragen, die gestellt und beantwortet werden müssen:

passiv: Woraus besteht ein Kundenauftrag?
Welche Positionen enthält eine Lieferliste (Ware)?
Welche Daten enthält eine Rechnung?
•••

aktiv: Welche Aktionen sind mit Rechnungsstellung verknüpft?
Welche Aktionen müssen ausgelöst werden, wenn Kundenauftrag kommt?

Neben dem Kontext müssen die internen Vorgänge (**Daten und Kontrollflüsse**) erfaßt und dargestellt werden:



Danach sind noch die folgenden Fragen offen:

passiv: Welche Infos sendet die Warenannahme an die Buchhaltung?
Welche Infos sendet die Bestellannahme an das Lager u. die Buchhaltung?
•••
aktiv: Wann soll das Lager den Einkauf "triggern"?
Welche Aktionen müssen erfolgen, wenn Ware geliefert wird?

Durch die Fragen und die Bilder wird
- das System nach außen abgegrenzt und
- die Beziehungen innerhalb des Systems ermittelt.

Die Frage lautet überwiegend
was passiert, wird befördert, ist zu speichern, ist auszuliefern, ...
und nicht
wie wird es realisiert!

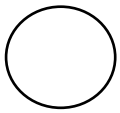
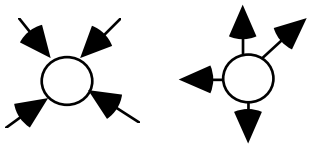
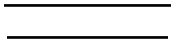
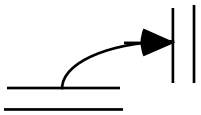
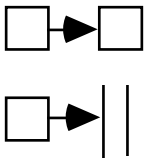
Benötigt werden Beschreibungsverfahren, mit denen das System (*Weinhandlung*) mit seinen
- Informationsflüssen und Datenstrukturen sowie den internen
- Abläufen beschrieben werden kann.

Die Beschreibung soll unabhängig von der späteren Realisierung erfolgen und *ist die Basis für den anschließenden Entwurf!*

Die Grundlage des Beschreibungsverfahrens in der Strukturierten Analyse bilden die Datenflußdiagramme, die hier bereits in Ansätzen verwendet wurden. Sie werden nun im Folgenden formal eingeführt.

4.2 Datenflußdiagramme

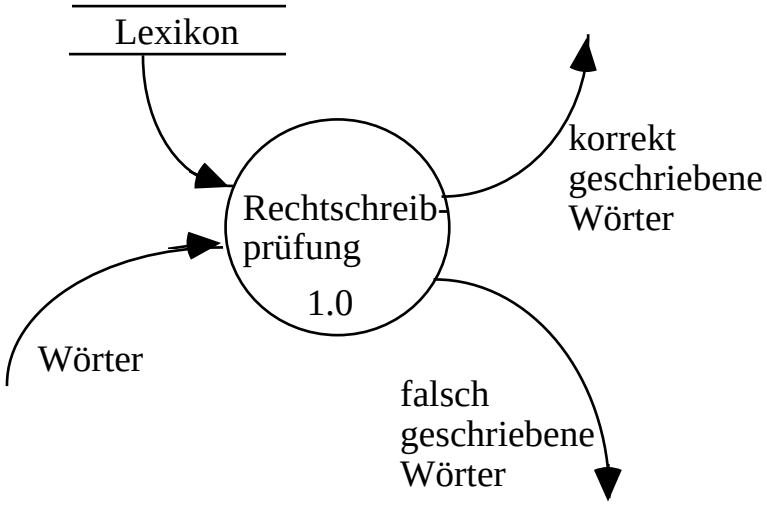
Datenflußdiagramme sind eine grafische Darstellung von Systemfunktionen (Prozessen) und ihren Schnittstellen, den Datenflüssen. Datenflußdiagramme können aus den folgenden Komponenten aufgebaut werden:

Symbol	Bedeutung	***nicht erlaubt***
	Datenfluß, Materialfluß	
	Aktion, Prozeß Aufgabe	
	Speicher, Ablage	
	Quelle oder Senke von Information oder Material	

Mit ihnen lassen sich die **funktionale Struktur**
- bereits bestehender (Analyse), wie auch
- neu zu entwickelnder Systeme (Spezifikation u. abstraktes Design)

darstellen.

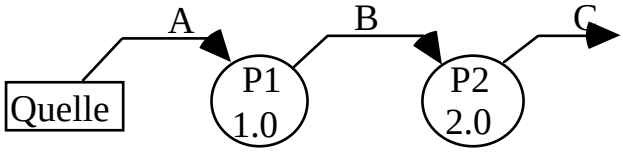
Beispiel:



Unter der funktionalen Struktur wird hier

> die Hierarchie von Systemaktivitäten zusammen mit ihren Schnittstellen untereinander, sowie zur Umgebung des Systems verstanden.

Funktional heißt in diesem Falle statisch, d.h. es ist kein zeitlicher Ablauf enthalten!



Nach De Marco ist P_i unendlich schnell! Aber es ist klar, daß B erst nach A berechnet werden kann.

Datenflußdiagramme zeigen anschaulich

- welche Daten auf welchen Wegen durch das System fließen,
- welche Bearbeitungsstationen sie dabei passieren und von
- welcher Art die jeweils vorgenommene Bearbeitung/Transformation ist.

Es werden die Ein-/Ausgabedaten, d.h. die Schnittstellen, für das gesamte System ebenso wie für jeden enthaltenen Bearbeitungsprozeß deutlich.

\ Aktivitäten: können manuell oder maschinell ausgeführt werden,
\ über technische Realisierung erfolgt keine Aussage!

4.2.1 Prozesse

Ein Prozess, auch Verarbeitungsknoten, im Englischen: node oder bubble, genannt, stellt eine Aktivität dar, die ihre einfließenden Daten in ausfließende Daten transformiert. Wie diese Transformation erfolgt, wird entweder durch die grafische Zerlegung in Teilkomponenten (Verfeinerungsdiagramm des Prozesses), durch eine Entscheidungstabelle oder durch eine textuelle Spezifikation beschrieben.

Für Prozesse gelten die folgenden Regeln:

- Jeder Knoten besitzt einen für seine Funktion charakteristischen Namen. Der Name sollte aus einem Verb und einem Substantiv bestehen.

- Die Benennung reicht in der Regel nicht aus, um den Zweck eines Knotens genau zu beschreiben. Die Funktion eines Knotens wird daher durch eine Prozeßspezifikation (P-Spec, Minispec) ausführlich beschrieben.

- Jeder Knoten besitzt eine Nummer, die ihn innerhalb des Systemmodells eindeutig identifiziert. Das Nummerierungsschema wird weiter unten erläutert.

Wie bereits erwähnt, wird die Ablaufsteuerung eines Systems nicht durch Knoten beschrieben, dies erfolgt mit den Mitteln der SA/RT, die in einem späteren Kapitel eingeführt werden. Im Rahmen der SA ToDo

4.2.2 Datenflüsse

Ein Datenfluss ist vergleichbar mit einer Pipelin, durch den Daten oder Material transportiert werden. Er kann aus mehreren Komponenten bestehen.

Ein einfacher Datenfluss wird durch eine durchgezogene Linie zwischen zwei Knoten dargestellt. (Bei einem Kontrollfluss, der erst bei SA/RT zum Tragen kommt, ist es eine gestrichelte Linie.) Die Flussrichtung wird durch einen Pfeil an einem Ende der Linie symbolisiert. Bidirektionale Flüsse, die also sowohl eingehen als auch ausgehen, haben an beiden Enden einen Pfeil.

Alle Datenflüsse müssen benannt und im Data-Dictionary (DD) eingetragen werden. Jeder Datenfluss erhält einen eindeutigen und sinnvollen Namen. Der Name ist i.A. ein Substantiv. Will man zusätzlich noch bestimmte Eigenschaften von ihnen beschreiben, so wird das Substantiv durch ein Adjektiv ergänzt.

Die Linie zerfällt in zwei Teile, ein Anfangs- und ein Endsegment, die durch einen kleinen Kreis (Flussknoten) voneinander getrennt sind. Normale Datenflüssen haben nur einen Namen, der für beide Segmente gilt. Der Einfachheit halber wird dann der Flussknoten häufig weggelassen.

Ein Fluss kann aber auch verfeinert oder aufgeteilt (Split-Fluss) werden oder mehrere Flüsse können zu einem zusammengeführt (Merge-Fluss) werden.

- Bei der Verfeinerung eines Flusses erhält jedes Segment einen eigenen Namen. Verfeinerung bedeutet, dass ein Segment ein Teil des anderen ist.

- Mit Split-Fluss wird ein Fluss bezeichnet, der nur ein Anfangssegment, aber mehrere Endsegmente hat. Dabei ist es unerheblich, ob diese Endsegmente Verfeinerungen des Flusses sind oder der gesamte Fluss selbst in mehrere Richtungen weiterfließt.

- Mit Merge-Fluss wird ein Fluss bezeichnet, der mehrere Anfangssegmente, aber nur ein Endsegment hat. Dabei ist es unerheblich ob diese Anfangssegmente Verfeinerungen des Flusses sind oder der gesamte Fluss selbst aus mehreren Richtungen kommt.

Die Verfeinerung, das Aufspalten und das Verbinden von Flüssen muss über die Definition der Flüsse (Segmentnamen) über das Data Dictionary abgesichert sein. Darauf wird später noch ausführlich eingegangen.

4.2.3 Speicher

Ein Speicher dient zur Ablage der auf den einfließenden Flüssen transportierten Daten. Prozesse können lesend und schreibend auf einen Speicher zugreifen. Ein Speicher trägt immer den Namen des Typs der in ihm enthaltenen Information, d.h. den Namen der mit ihm verbundenen Flüsse.

Für Speicher gelten die folgenden Regeln:

- Jeder Speicher erhält einen sinnvollen Namen, der einen Rückschluss auf seinen Speicherinhalt ermöglicht. Der Name muss im Data Dictionary definiert werden.

- Ein Speicher sollte nur dann angelegt werden, wenn mindestens zwei Knoten zu unterschiedlichen Zeitpunkten auf den Inhalt zugreifen.

- Datenflüsse zwischen Speicher und Knoten können unbenannt sein. In diesem Fall wird auf den gesamten Speicher zugegriffen.

Den Flüssen von oder zu Speichern können generell keine Namen zugewiesen werden, da sie wie oben erwähnt mit dem Speicher, mit dem sie verbunden sind, identifiziert werden. Es sind jedoch Verfeinerungen auf der dem Speicher abgewandten Seite eines Flusses zulässig, die dann auch benannt werden können. Hiermit ist es möglich auszudrücken, dass nicht der gesamte Speicher gelesen/geschrieben wird, sondern nur Teile davon.

Da Speicher Teil des Systems sind, können sie auf der Kontextebene (siehe unten) nicht erscheinen. Wenn sie nicht Teil des Systems sind, dann sind sie nicht als Speicher, sondern als Terminator zu modellieren.

4.2.4 Terminatoren

Ein Terminator dient als Schnittstelle zur Umgebung des Systems. Er kann Ausgangs- (Quelle) oder Endpunkt (Senke) sowohl von Daten- als auch von Kontrollflüssen sein. Es kann sich um externe Systeme, um technische Prozesse,

um handelnde Personen etc. handeln, die eben nicht Bestandteil des zu modellierenden Systems sind. In diesem Sinne kann es sich auch um eine benachbarte Teilkomponente handeln, wenn gerade nicht das ganze System sondern nur eine Teilkomponente modelliert wird.

Es gelten folgende Regeln:

- Terminatoren werden mit einem Substantiv benannt.

- Terminatoren dürfen nur auf der Kontextebene vorkommen, siehe unten.

Zwischen Terminatoren direkt dürfen keine Datenflüsse gezeichnet werden. Das würde ja bedeuten, dass ein solcher Datenfluss am System vorbeifließt, also auch mit dem System nichts zu tun hat. Wenn das aber so ist, dann gehört der Fluß nicht in das Diagramm, mit dem wir ja das System modellieren wollen.

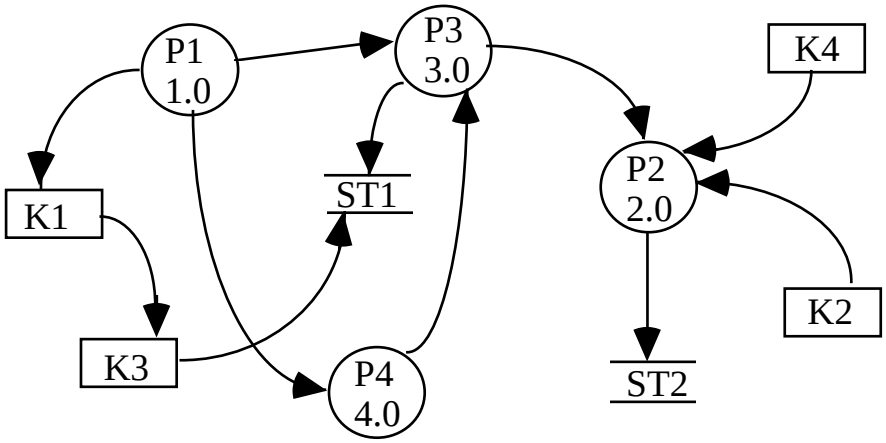
4.2.5 Konsistenzbedinungen

Insgesamt gelten für die Verbindung der Komponenten eines Datenflussdiagramms gewisse Konsistenzbedingungen:

- Flüsse dürfen nicht von Speicher zu Speicher oder von Quelle zu Senke fliessen, auch nicht von einer Quelle direkt zu einem Speicher oder aus einem Speicher direkt in eine Senke.
- Zyklische Datenflüsse sind nicht erlaubt.
- Prozesse oder Speicher dürfen nicht nur Eingänge oder nur Ausgänge haben (aber: Timer).

Letztere Regel gilt allerdings bei Speichern in Unterdiagrammen nicht: da ein Unterdiagramm nur ein Teil modelliert, kann es vorkommen, dass in einem Diagramm nur ein Lesezugriff auf einen Speicher zu sehen ist und in einem anderen der Schreibzugriff. Auf die Regeln wird später noch einmal genauer eingegangen.

Das folgende Diagramm enthält eine Reihe von Verletzungen dieser Konsistenzbedingungen:



4.3 Beispiel zur Erstellung eines Datenflußdiagrammes

Folgende verbale Problembeschreibung sei gegeben:

> Der Kunde gibt an der Gepäckannahme das Gepäck und seine Reisepapiere ab. Der Schalterbeamte erkennt daraus den Zielort, entnimmt einer Schublade einen Gepäckschein für jedes Gepäckstück und füllt die Scheine aus. Er trennt die Scheine und heftet je einen Teil an das Gepäckstück und legt es auf das Förderband. Er gibt dem Kunden seine Reisepapiere incl. der Kontrollabschnitte zurück.

? Was soll automatisiert werden ?

| *Gepäckannahme*

Welches sind die Schnittstellen zum System, d.h.:

? Welche Daten- / Materialflüsse treten auf ?

| Gepäck, Reisepapiere, Gepäckschein, Kontrollabschnitte, gekennzeichnetes.Gepäck (oder Gepäck+Schein)

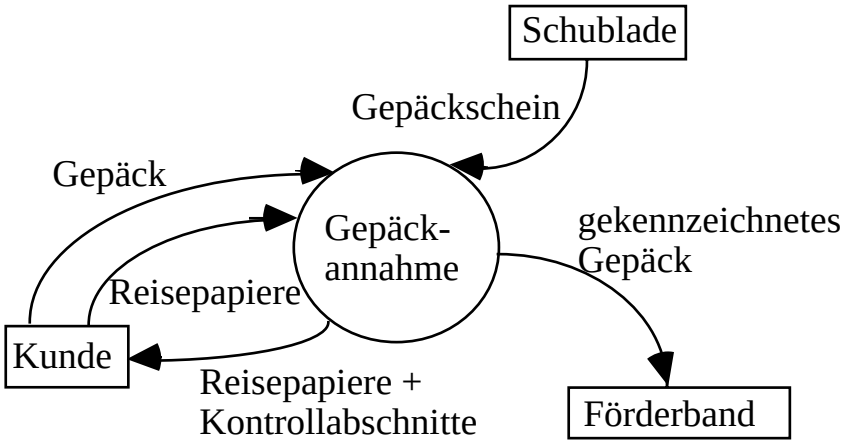
? Welche Daten- / Material-Quellen / Senken sind vorhanden ?

| Kunde | Schublade | Förderband

Hilfsmittel:
Substantive (∏ Quelle, Senke, Daten, Material, Speicher...) und
Verben (∏ Tätigkeiten, Prozesse, ...)
fett/bunt markieren !

Der Kunde gibt an der *Gepäckannahme* das Gepäck und seine Reisepapiere ab. Der Schalterbeamte erkennt daraus den Zielort, entnimmt einer Schublade einen Gepäckschein für jedes Gepäckstück und füllt die Scheine aus. Er trennt die Scheine und heftet je einen Teil an das Gepäckstück und legt es auf das Förderband . Er gibt dem Kunden seine Reisepapiere incl. der Kontrollabschnitte zurück.

Resultierendes Datenflußdiagramm:



4.4 Hierarchien von Datenflußdiagrammen

Da alle nicht trivialen Systeme zu komplex sind, um sie in einem DFD hinreichend genau darzustellen, ist eine Einteilung in Abstraktionsebenen zwingend erforderlich. Zur Beschreibung solcher Systeme werden hierarchische Modelle gebildet, die auf den verschiedenen Abstraktionsebenen jeweils genauere Beschreibungen erlauben.

4.4.1 Verfeinerung eines Modells

Durch die iterative Top-Down Vorgehensweise erhält man eine Hierarchie von DFDs. Die Ebenen sind die Abstraktionsebenen des Modells.

Die oberste Ebene (Ebene 0) wird gebildet vom **Kontextdiagramm,** es besteht nur

- aus dem **Superprozeß** sowie
- allen Verbindungen zu externen Quellen und Senken (Terminatoren).

Externe Quellen und Senken bilden die Verbindung des Systems zur Außenwelt. Sie bescheiben somit **alle** Ein- und Ausgaben! Quellen und Senken existieren nur auf der Ebene 0. Das obige DFD zur Gepäckannahme ist ein Beispiel für ein Kontextdiagramm.

Das Diagramm der Ebene 1 enthält eine Zerlegung des Systems in die Hauptkomponenten. Es heißt deswegen auch **Systemdiagramm** oder **Übersichtsdiagramm**.

Weitere Ebenen verfeinern bei Bedarf das Systemmodell.

Ein Diagram, in dem ein Knoten verfeinert wird, steht mit der Verfeinerung in einer Vater-Sohn-Beziehung (Parent-Child). Der Knoten, der verfeinert wird, heißt Vaterknoten. Das Sohn-Diagramm erhält den Namen des Vaterknotens.

- Jeder Knoten erhält entsprechend seiner Position eine eindeutige per Punkt strukturierte Nummer und einen Namen.

- Alle ein- bzw. ausgehenden Datenflüsse eines Vaterknotens sind auch ein- bzw. ausgehende Datenflüsse des Sohn-Diagramms (balancing).

- Datenspeicher im Vater-DFD werden stets auch im Sohn-DFD eingetragen und dort mit mindestes einem der Knoten verbunden. D.h. erfolgt vom Vaterknoten ein Zugriff auf einen Datenspeicher, so ist dieser auch im Sohn-Diagramm erkennbar.

Die Verfeinerung erfolgt solange, bis eine Stufe erreicht ist, in der die Prozesse übersichtlich sind und durch eine Prozeßspezifikation beschrieben werden können. Diese Prozesse heißen elementare Prozesse. Das sind also solche Prozesse, die nicht weiter verfeinert werden.

Das Nummerierungsschema für die Prozesse lautet folgendermaßen:

- Der Prozess im Kontextdiagramm (System- oder Superprozess) erhält die Nummer 0.
- Die Prozesse im Übersichtsdiagramm werden durchnummeriert, beginnend mit der Nummer 1.
- Die Nummern aller anderen Knoten setzen sich zusammen aus der Nummer des Vaterknotens, einem Punkt und einer diagramm-lokalen Nummerierung.
- Bei der diagramm-lokalen Nummerierung werden die Knoten des Diagramms durchnummeriert, beginnend mit der Nummer 1.

Die Top-Down Vorgehensweise ist nicht zwingend. Bei größeren Projekten wird häufig mit einer tieferen Ebene begonnen. Zu einem späteren Zeitpunkt werden dann die darüberliegenden Abstraktionsebenen ergänzt.

4.4.2 Beispiel: Systemdiagramm der Gepäckannahme

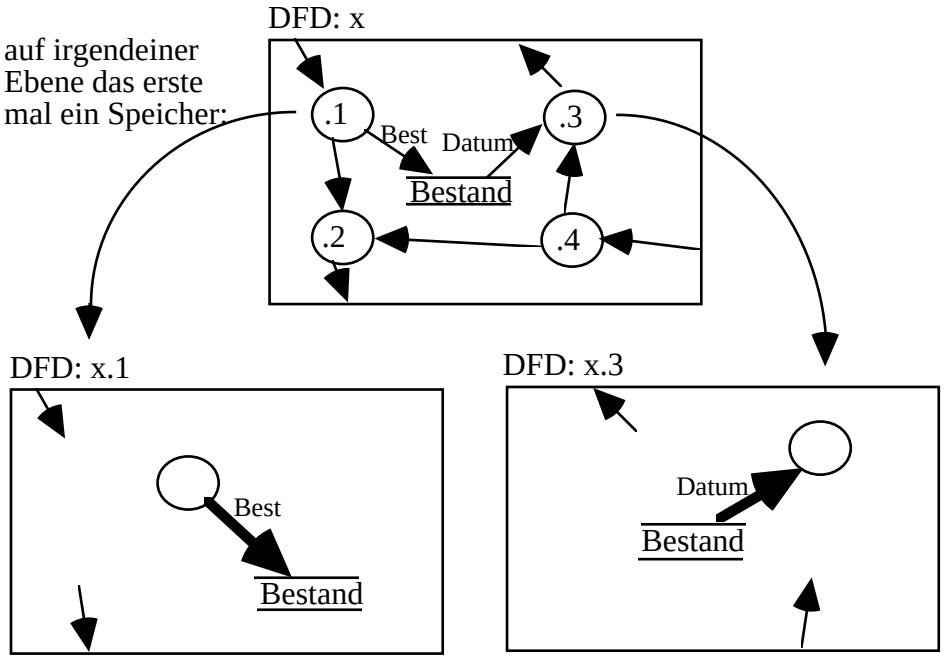
(ToDo)

4.4.3 Balancing zwischen den Diagrammen

ToDo: Horizontales Balancing

Vertikales Balancing: Alle ein- bzw. ausgehende Datenflüsse des Vaterknotens sind auch ein- bzw. ausgehende Datenflüsse des Sohndiagramms. Erfolgt vom Vaterknoten ein Zugriff auf einen Datenspeicher, so muss dieser Zugriff auch im Sohn-Diagramm erkennbar sein.

Beispiel:



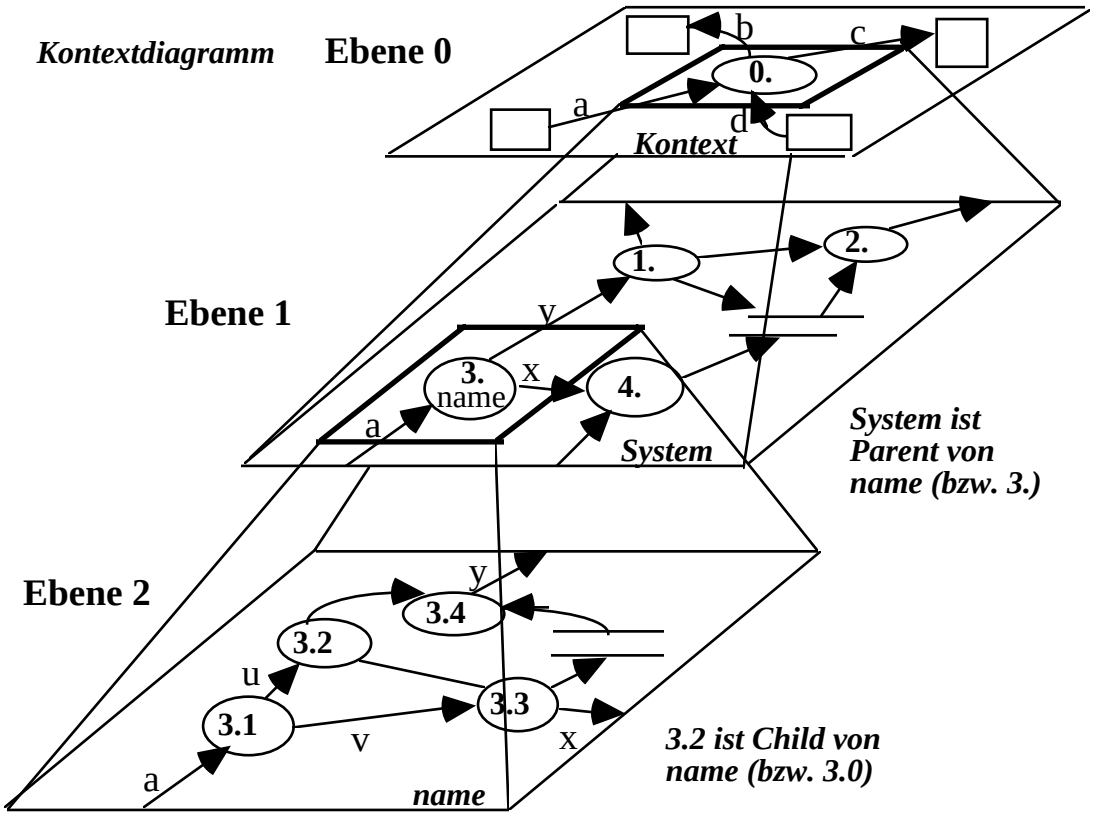
Die Verbindung der Speicher erfolgt nur über gleichlautende Namen! Durch diese Regel kann es wie im Beispiel zu Diagrammen kommen, in denen auf Speicher nur lesend bzw. nur schreibend zugegriffen wird.

Die Regeln des Vertikalen Balancing gelten auch in Verbindung mit dem Datenlexikon und werden dann als Datenlexikon-Balancing bezeichnet. Aufgrund der gezeigten Datenlexikon-Einträge sind auch folgende Diagramme korrekt balanciert.

ToDo

4.4.4 Schematische Darstellung der Hierarchie-Ebenen

Im folgenden Bild ist noch einmal zusammenfassend das Schema der Hierarchien von Diagrammen über drei Ebenen zu sehen.



Für alle Prozesse kann und sollte eine Prozeßbeschreibung, zumindest in der Form einer Minispec, erstellt. werden. Für alle Prozesse, die nicht weiter verfeinert werden muss eine Prozessbeschreibung erstellt werden. Prozeßbeschreibungen werden nun im Folgenden besprochen.

4.5 Prozeßspezifikationen

Jeder Prozeß kann durch eine **Prozeßspezifikation (P-Spec)** beschrieben werden. Elementare Prozesse, das sind Prozesse, die nicht verfeinert werden, müssen durch eine Prozeßspezifikation beschrieben werden. Statt P-Spec wird häufig auch der Begriff **Minispec** benutzt.

Eine Prozeßspezifikation kann textueller Art sein. Sie kann dann abgefaßt sein:

1. **natürliche Sprache (lediglich Kommentar)**
2. **strukturierte Sprache (Pseudo-Code)**

Eine Prozeßspezifikation kann aber auch formal abgefaßt sein, dann hat sie die Form einer

3. **Entscheidungstabelle**

Diese Möglichkeiten werden im folgenden aufgezeigt:

4.5.1 Textuelle Prozeßspezifikationen

ToDo: Form, E/A im Text referieren

Zu 1: Kommentar

Process:		3.4.2
Name:		Beton mischen
Input:		Zement, Sand, Wasser
Output:		Beton, Fertig
Body:	*	Hier steht unser Herr Müller und mischt zuerst den "**Sand**" mit dem "**Zement**" gut durch. Dann mischt er unter Zugabe von "**Wasser**" nochmals, bis ein dicker Brei entstanden ist. Wenn der "**Beton**" fertig ist, gibt er das Signal " **Fertig**" aus.
	*	

Zu 2: Pseudo-Code

Process:	3.4.2
Name:	Beton mischen
Input:	Zement, Sand, Wasser
Output:	Beton, Fertig
Body:	repeat
	Zement und **Sand** mischen
	until gut gemischt
	repeat
	Wasser zugeben und mischen
	until dicker Brei entstanden
	Beton = dicker Brei
	Fertig = true.

4.5.2 Formale Prozeßspezifikationen

Zu 3) Entscheidungstabelle

Bedingung	Regel 1	Regel 2	Regel 3	Regel 4
Kunde hat Fahrkarten	N	J	J	J
Kunde hat Gepäck	-	N	J	J
Gepäck schwerer als 50 Kg	-	N	N	J
Kunde zu Fahrkartenausgabe schicken	X			
Abfertigen		X	X	X
Gepäck aufgeben				X
Nachfragen: Wo ist Gepäck ?		X	X	

Hier **nur binäre** Eingänge, d.h J/N. ToDo!!

4.6 Das Datenlexikon

Für alle im Verlauf der Analyse / Sollspezifikation vergebenen Bezeichner ist ein definierender Eintrag in einem Katalog vorzunehmen, dem sogenannten **Datenlexikon** (für Data Dictionary, abgekürzt: **DD**). Dies

- legt die Verwendung eines Bezeichners fest,
- vermeidet die mehrmalige Vergabe desselben Bezeichners, was bei mittleren Projekten bereits zu Problemen führt,
- legt die Komponenten eines Datenflusses fest,
- legt die Objekte fest, welche sich in einem Speicher befinden dürfen,
- bildet eine zentrale Übersicht über vergebene Bezeichner,

und ist nicht nur bei Werkzeugunterstützung sinnvoll!

Anforderungen an ein DD:

> Keine redundante Datenhaltung (wegen Konsistenz).
>
> Datendefinition muß hierarchische Verfeinerung erlauben (Komplexität).
>
> Datendefinition muß bestimmter Syntax genügen (formal überprüfbar).

Das Datenlexikon ist somit ein zentraler Definitionskatalog für sämtliche Daten und Speicher. Es löst Verständigungsprobleme durch Festlegung der Struktur und der Bedeutung der Datenflüsse. Die meisten Worte - und seien sie noch so alltäglich - sind so vieldeutig, dass sie zur Spezifikation nur dann herangezogen werden können, wenn ihre Bedeutung vorher festgelegt worden ist.

4.6.1 Einträge in ein Data Dictionary

Einträge in das Data Dictionary müssen einer bestimmten Syntax genügen, die an die erweiterte Backus-Naur-Form angelehnt ist. Die Syntax gestattet es, Daten- und Kontrolltypen aus anderen Typen zusammenzusetzen. Allerdings: Datentypen dürfen nicht aus Kontroll- und Kontrolltypen dürfen nicht aus Datentypen zusammengesetzt werden.

Neben den zusammengesetzten Typen können auch terminale (primitive) Typen definiert werden, die sogenannten Literals. Primitive Typen können nicht weiter verfeinert werden. Sie können beispielsweise repräsentieren:

- diskrete Daten

(mit Wertebereich "true", "false" oder "rot", "grün",...)

- kontinuierliche Daten

(falls aus Kontext dann oft mit Bereich: -5 bis +20 ^{o}C,...)

Jeder im Data Dictionary definierte Typ soll, wenn auch über mehrere Ebenen hinweg, letztlich aus Literals zusammengesetzt sein.

Ein Eintrag im Data Dictionary heißt Typdefinition (type definition). Ein solcher Eintrag erfolgt in einer Form, die an die Erweiterte Backus-Naur-Form (EBNF) angelehnt ist:

<Name des DD-Eintrags> = <Datendefinition>.
= heißt hier: *besteht aus den Komponenten*

<Datendefinition> beschreibt den Aufbau per:

Symbol	Semantik
"...."	terminaler Datenwert (z.B. "TRUE", "gelb")
+	Sequenz von Objekten
( )	optional vorhandener Teil (entspricht {...}1)
[a | b]	Alternative: entweder a oder b (auch mehr als 2)
u{ }o	Wiederholung min. u mal bis max. o mal

Eine Sequenz (sequence) gestattet es, einen Typ aus mehreren Komponenten verschiedener Typen zusammenzusetzen.

Eckige Klammern um eine Aufzählung (enumeration) von Typen kennzeichnen eine Alternative. Die einzelnen Alternativen, die aufgezählt werden, werden durch einen senkrechten Strich getrennt. Jedes Objekt des definierten Typs enthält eine Komponente eines beliebig ausgewählten Typs der Aufzählung.

Ein Typname in runden Klammern innerhalb einer Typdefinition beschreibt, dass eine Komponente dieses Typs an der Zusammensetzung des definierten Typs beteiligt sein kann, aber nicht muss. Die Komponente ist also optional.

Ein Typname in geschweiften Klammern beschreibt, dass Komponenten dieses Typs mehrfach an der Zusammensetzung des definierten Typs beteiligt sind (iteration). Die Anzahl der Iterationen kann durch Indizes bestimmt werden.

Beispiele für DD-Einträge sind:

```
Name       = Vorname + Nachname.
Datum      = Tag + Monat + (Jahr).
Zeichen    = [ Buchstabe | Ziffer ].
```

```
Namensfeld = 1{Zeichen}30.
```

Alle Datenflüsse und Speicher, die in einem DFD gezeigt werden, müssen im DD definiert werden. Zusätzlich können auch Komponenten von Speichern und Datenflüsse im DD definiert sein, die nicht in Diagrammen auftreten. So könnte im obigen Beispiel zwar Namensfeld in einem DFD auftreten, seine Komponente, nämlich Zeichen möglicherweise nicht. Wenn ein Typname zur Definition eines anderen Typs verwendet wird, dann muss er auch im DD definiert werden.

An beliebigen Stellen auf der rechten Seite einer Definition kann per

```
* <text> *
```

Kommentar eingestreut werden, um den Sachverhalt zu verdeutlichen.

4.6.2 Einfaches Beispiel

Die Definition von deutschen Kfz-Kennzeichen lautet etwa:

> Zuerst kommen 1 bis 3 Buchstaben gefolgt von einem Strich.
> Danach können optional 1 bis 2 Buchstaben stehen.
> Durch einen Zwischenraum getrennt folgen am Ende dann noch 1 bis 4 Ziffern.

In der oben beschriebenen EBNF wird das folgendermaßen formalisiert:

```
KFZKZ  =    1{ BU }3 + "-" +
            ( 1{ BU }2 +)
            " " + 1{ ZI }4.
```

Ein vollständiges DD setzt natürlich voraus, daß alle in einer EBNF-Definition neu verwendeten Namen ebenfalls im DD definiert werden. In unserem Beispiel müßte man also noch definieren, was Buchstaben und was Ziffern sind.

```
BU     =    [ "A" | "B" |  .... | "Z" ].
ZI     =    [ "0" | "1" |  .... | "9" ].
```

Dadurch ist beispielsweise jetzt festgelegt, daß nur Großbuchstaben auftreten dürfen.

Über die Möglichkeit, Terminale an beliebiger Stelle einzuführen um die Beschreibung abzubrechen, hat man die Möglichkeit, das DD übersichtlich zu halten. Am Anfang wird man relativ früh Terminale einsetzen, während man mit dem Fortschritt der Analyse die Erkenntnisse über einen Typ durch eine Verfeinerung in das DD mit aufnehmen kann.

Das obige Beispiel kann beispielsweise auch folgendermassen aussehen:

```
KFZKZ  =    1{ BU }3 + STRICH +
            ( 1{ BU }2 ) +
```

```
                LEER + 1{ ZI }4.

BU     =    "Grossbuchstaben".
ZI     =    "Ziffern".
LEER   =    "Leerzeichen".
STRICH =    "Bindestrich".
```

Damit sollte klar sein: es gibt viele Möglichkeiten die Einträge im DD zu gestalten. Man muss dabei allerdings bedenken, dass immer dann wenn Terminal auftauchen, ein Werkzeug nicht mehr weiter prüfen kann, d.h. der Text zwischen den Apostrophen kann und wird nicht analysiert. Die anderen Einträge können jedoch auf Vollständigkeit und Konsistenz geprüft werden.

4.6.3 Weitere Beispiele für Einträge

Beispiel 1: Auftragsverwaltung

Anzahl =	“Ganzzahl“.
Artikel =	“ “ * max. 30 Zeichen langer String *.
Auftrag =	Auftragsdatum + 1{Position}100 + (Liefertermin) + [Kundenname | Kundennummer].
Auftragsdatum =	Monat + Tag + (Jahr).
Auftragskartei =	Kopf + 1{ Auftrag }1000.
BestellNr =	“ “ * 5 -stellige Zahl, mit folgender Bedeutung 1. Ziffer: 0 für Standardartikel 1 für ... *.
Kunde =	Kundenname + Adresse + Kundennummer.
Kundenkartei =	1{Kunde}5000.
Kundenname =	Nachname + (Vorname).
Kundennummer =	“ “ * 10 stellige Zahl , wie folgt aufgebaut: .... *.
Position =	Anzahl + BestellNr + Artikel + Einzelpreis + Positionspreis.

Beispiel 2 Literatur Datenbank

```
Buch =          Kennung + T1 +
                Typ + T2 +
                1{Verfasser}4 +
                Titel + T5 +
                ( Jahr + T7 ) +
                ( ISBN + T8) +
                Abstract + T9 +
                Bemerkung.

Kennung =       1{ char }8 .

Typ =           ["Buch" | "Zeitschrift" | "Diplomarbeit" | "Skript" ].

Verfasser =     Nachname + T3 +
                (Vorname + T4).

Titel =         1{ char }80.

Jahr =          ["1" | "2"]  + Ziffer + Ziffer + Ziffer.

ISBN =          Ziffer + "-" +
                3{ Ziffer }3 + "-" +
                4{ Ziffer }5 + "-" +
                Ziffer.

Abstract =      1{ char }400.

Bemerkung =     1{ char }20.

LiteraturDB =   1{ Buch }5000.
```

4.7 Vorgehensweisen bei der Strukturierten Analyse

Man unterscheidet drei unterschiedliche Vorgehensweisen:

- **datenflußorientiert**,
- **funktionsorientiert** und
- **ereignisorientiert**.

\+ Datenflußorientierter Ansatz

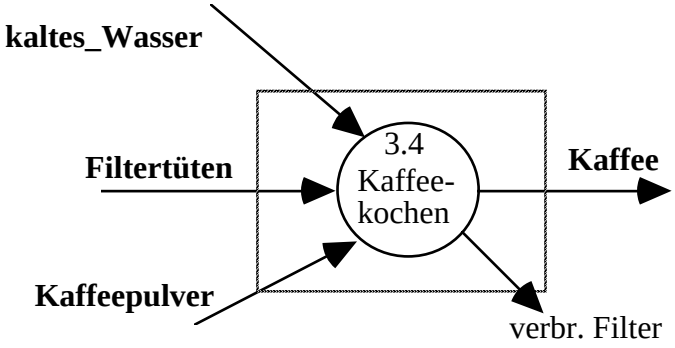
Die Funktionen ergeben sich aus den primär betrachteten Daten- und Materialflüssen.

Zuerst Fragen der Form:

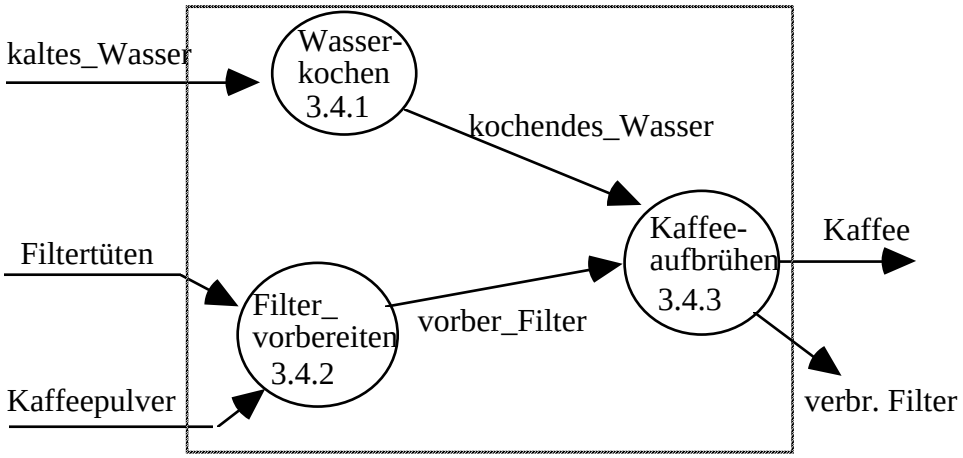
welche **Objekte** werden bewegt ?

Jedes wird mit Namen versehen und ins Data Dictionary (DD) aufgenommen.

kaltes_Wasser
Filtertüten
Kaffeepulver !
Kaffee
verbr. Filter



Dann Verfeinerung der Aktion Kaffeekochen (Knoten 3.4):



+ Funktionsorientierter Ansatz:

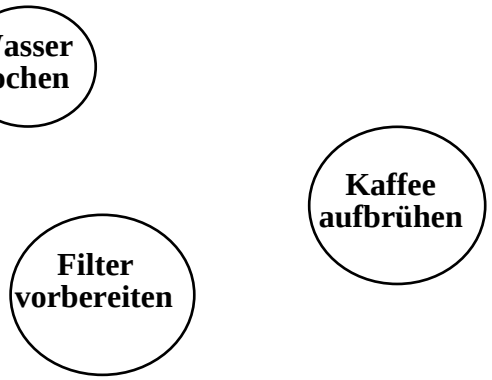
Zuerst werden die Aktionen und Tätigkeiten (Funktionen) zusammengestellt, entsprechende Fragen:

welche **Aktionen, Handlungen, Prozesse** treten im System auf ?

Wasser kochen

Filter vorbereiten !

Kaffee aufbrühen



Das Datenflußdiagramm entsteht durch die Verbindung der Funktionen durch geeignete Daten- / Materialflüsse.

+ Ereignisorientierter Ansatz:

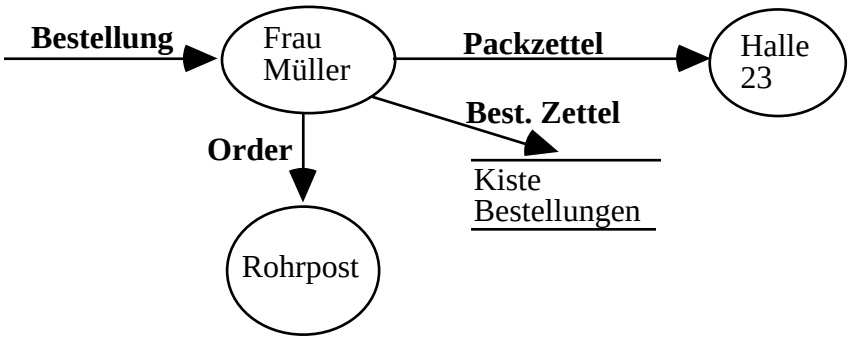
Der ereignisorientierte Ansatz kann erst mit den Mitteln von SA/RT durchgeführt werden, die im nächsten Kapitel vorgestellt werden.

4.8 Teilschritte der Strukturierten Analyse

Tom DeMarco [DeM 79] unterscheidet sieben Teilschritte der strukturierten Analyse:

1 Istanalyse

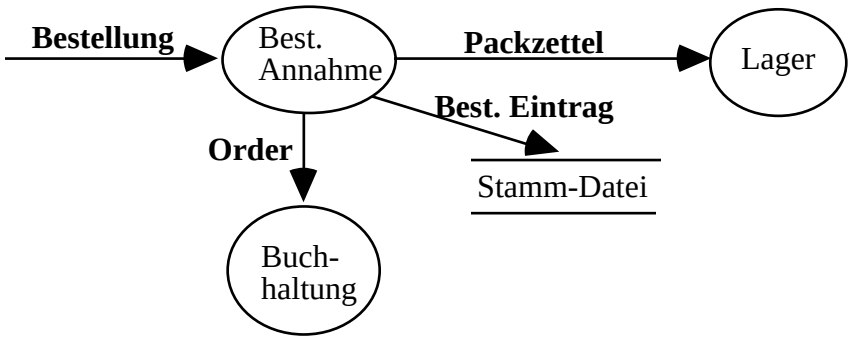
Das bereits vorliegende (alte) System bildet die Grundlage für ein neues. Darum Istzustand analysieren und als **physischen Istzustand** in einem **DFD** festhalten.



Problem:
Nicht algorithmisch denken und Ideen festhalten (d.h. **wie** also die Daten in einem Prozeß verändert werden), sondern **nur deklarative Spezifikation** aus der Sicht der relevanten Daten.

2 Entwurf einer logischen Sicht

Aus dem physischen Istzustand wird durch Abstraktion der logische Istzustand (ebenfalls in Form eines DFD) abgeleitet.



3 Logische Spezifikation des neuen Systems

Die geforderten Verbesserungen, Umstellungen, Erweiterungen, ... werden im neuen System auf der logischen Ebene eingearbeitet.
Es entsteht das **Sollkonzept** in Form eines logischen (abstrakten) DFD.

4 Bestimmung der physischen Eigenschaften

Das System wird in automatisierbare und manuelle Bestandteile (Aktionen, Prozesse,...) zerlegt.

Ergebnis: **physische(s) Sollkonzept(e)** bzw. DFD.

Frau Müller bleibt also am Telefon

Alternativen:
- *Bestellungen nur noch per Fax und OCR-SW*
- *maschinenlesbare Bestell-Vordrucke*
- *Spracherkennung und Voicemail-System.*

D.h. auch hier können mehrere Alternativen aufgezeigt werden.

5 Kosten/Nutzen- Analyse

Wirtschaftliche Kalkulation, evtl. mehrerer Alternativen.

6 Auswahl

Auch unter Berücksichtigung von
- personellen, infrastrukturellen,
- arbeitsphysiologischen, ergonomischen, ... Überlegungen

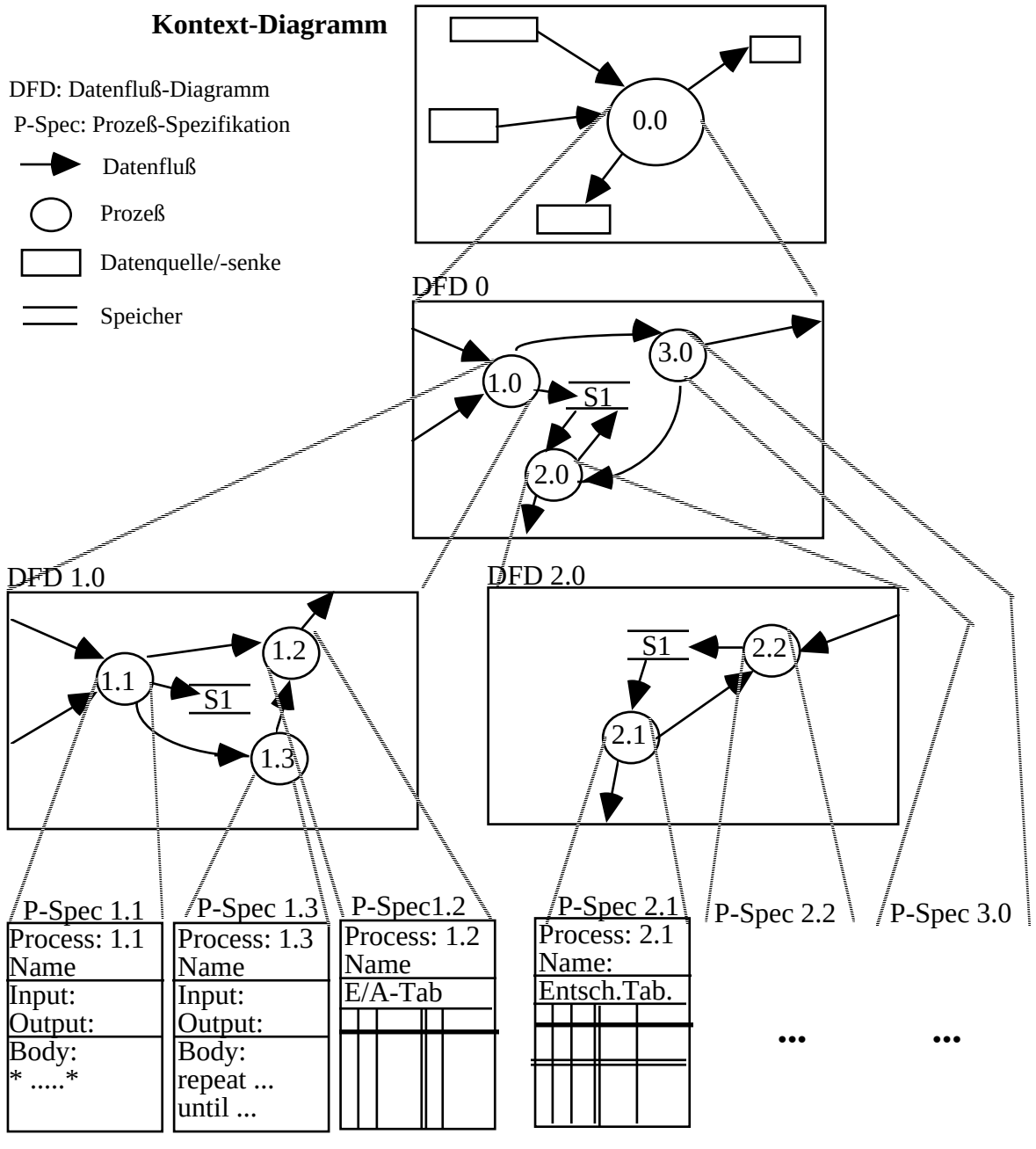
7 Dokumentation

Alle Dokumente werden in der Strukturierten Sollspezifikation zusammengefaßt (Teil des Pflichtenhefts).

Auf Schritt 1 und 2 kann verzichtet werden, wenn das logische Datenmodell direkt spezifiziert werden kann. Dies ist z.B. der Fall, wenn kein altes System existiert, auf welches eine Analyse abgestützt werden kann. Ebenso, wenn das neu zu entwickelnde System sich bewußt nicht am alten orientieren soll.

4.9 Zusammenfassung

Das folgende Bild verdeutlicht nochmals alle Elemente der SA.



Data-Dictionary (DD):	**Definition aller Datenflüsse und Datenspeicher (auch Teile davon)**

Zur Modellierung eines Systems in der Strukturierten Analyse stehen 3 Beschreibungsmittel zur Verfügung:

1. Mehrstufige **Datenflussdiagramme** beschreiben das System in verschiedenen Abstraktionsstufen. Es gibt dabei verschiedene Aspekte für das Ende der Dekomposition:

   + die Größe einer P-Spec < als eine DIN-A4
   + Prozeß besitzt genau einen Eingang und einen Ausgang
   + DFD besitzen definierte n:1 Beziehung bzgl. Ein- und Ausgängen

   Wegen Übersichtlichkeit und Verständlichkeit, sollte dabei auch der Umfang eines Diagrammes und der Hierarchie beschränkt werden:

   + max. 5 ± 2 Prozesse / Diagramm
   + und nicht mehr Ebenen.

   Damit kann man bei ausgeglichenen Bäumen 3^3 = **27** bis 7^7 = **823.548** Prozesse darstellen.

2. **Minispezifikationen** beschreiben die Aufgaben der in den Datenflussdiagrammen eingesetzten Knoten.

3. Im **Data-Dictionary (DD)** werden alle Datenflüsse und Datenspeicher definiert und abgelegt. Damit wird ihre Struktur und ihre Bedeutung verdeutlicht.

5 Die Methode SA/RT

Die Ideen der SA sind aus den Erfahrungen bei der Analyse von kommerziellen Systemen, wie Banken, Versicherungen, Buchungssystemen,... hervorgegangen.

DFD beschreibt darum in der SA die Zusammenhänge eines Systems nur funktional, d.h. insbesondere mehr oder weniger statisch. Jeder Knoten symbolisiert eine Aktion oder Handlung, welche einen Eingabestrom in einen Ausgabestrom umwandelt.

Für den genauen Zeitpunkt, zu dem eine Umwandlung stattfinden soll, die Zeit, welche zur Bearbeitung benötigt wird oder für die Reihenfolge, in der Aktionen ausgeführt werden sollen, gibt es keinerlei Hinweise im DFD.

Vergleich mit Schaltnetz in der Elektronik !
jedoch kommen im DFD auch Rückkopplungen vor !

Oftmals ist es jedoch notwendig, auch diese Zusammenhänge in einem ähnlich einfachen Diagramm beschreiben zu können. Dazu wurde SA erweitert. Die erweiterte Methode SA/RT unterstützt die ereignisgesteuerte Vorgehensweise. Damit können Echtzeitsysteme, mit Zeitabhängigkeiten, vordefinierten Antwortzeiten oder Ereignisabhängigkeiten modelliert werden.

Die in der Anforderungsanalyse bzw. Sollspezifikation von DeMarco eingesetzten **DFD**, **P-Spec** und Einträge im **DD** wurden erweitert, um ereignisgesteuerte Probleme zu beschreiben. Hatley und Pirbhai [HaP88] erweiterten das DFD um zusätzliche Komponenten, die es erlauben, **Kontrollflüsse** (Control-Flow, CF) zu beschreiben. Die Verarbeitung von Kontrollflüssen wird in den sogenannten **Kontrollspezifikationen** beschrieben.

5.1 Kontrollflußdiagramme

Ein Kontrollfluß-Diagramm (**CFD**) enthält:

- **Prozesse**
- **Kontrollflüsse**
- **Speicher**
- **Kontrollspezifikationen (C-Spec)**

Die Prozesse sind identisch mit denen in den DFDs, d.h. sie tragen auch gleiche Namen und gleiche Nummern. Kontrollflüsse werden grafisch durch gestrichelte Pfeile dargestellt. Speicher können neben Verarbeitungsdaten nun auch Steuerdaten aus Kontrollflüssen aufnehmen und speichern. Kontrollspezifikationen sind neu und werden grafisch durch einen dicken Balken (Bar, Barren) dargestellt.

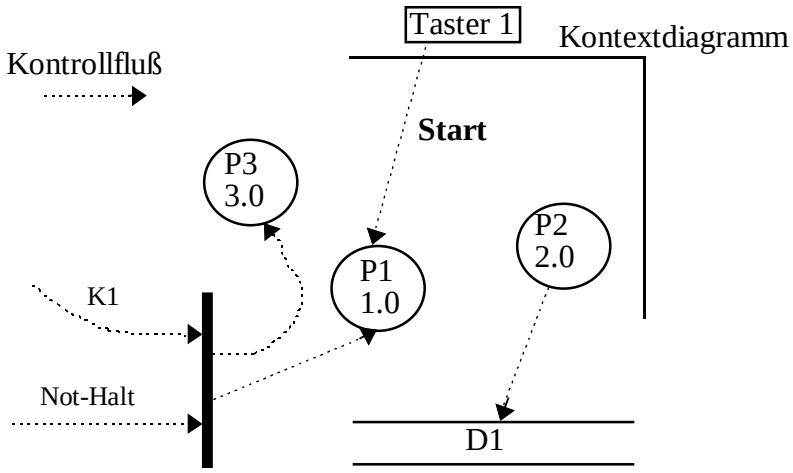
Ein CFD kann somit durchaus isolierte Prozesse enthalten, zu denen es weder ein- noch ausfliessende Kontrollflüsse gibt. Denn auch wenn ein Prozeß nur datenfluss-relevant ist, muß er ins CFD übernommen werden. Auf der anderen Seite könnte er durch eine Kontrollspezifikation gesteuert werden, was auch im Diagramm nicht sichtbar ist, und weiter unten erläutert wird.

5.1.1 Einfaches Beispiel

ToDo

5.1.2 Kontrollflüsse und ihre Verarbeitung

Kontrollflüsse treten zwar als Ein- / Ausgangsgrößen von Prozessen auf, werden jedoch nicht von ihnen verarbeitet, sondern lediglich weitergereicht. Kontrollflüsse werden einzig und allein in den Kontroll- oder Steuerprozessen verarbeitet, die durch einen Barren im Diagramm repräsentiert werden. Der Begriff Prozeß wird jedoch für die Barren selten verwendet, geläufiger ist der Begriff Kontrollspezifikation, oder Kontrollinstanz. Nur wenn ein Kontrollfluss in einen Barren hineinfließt, wird er dort verarbeitet.



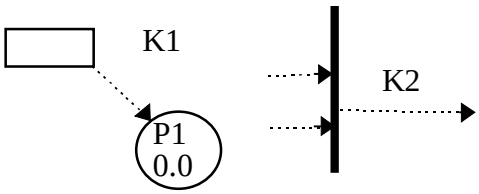
Kontrollflüsse können ebenso in Datenspeichern abgelegt werden, wie Datenflüsse.

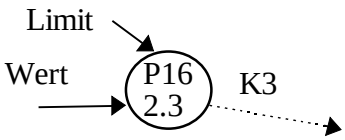
Prozesse dienen nicht der Verarbeitung von Kontrollflüssen oder zur Ablaufbeschreibung. Endet ein CF in einem Prozeß, so muß er in tiefere Ebenen verfolgt werden, bis er in einem **Barren** verarbeitet wird.

Im obigen Bild wird P1 nicht durch Taster 1 gestartet. Soll dies der Fall sein, so muß Start Eingang einer entsprechenden Prozeßaktivierungstabelle für P1 sein!

Wie im folgenden Bild zu sehen, haben Kontrollflüsse ihren Ursprung

- im **Kontext** (-Diagramm), d.h. kommen von außerhalb des Systems (Quelle),
- in einer **Kontrollspezifikation** oder
- in einem **Prozeß** und werden aus einem Datenfluß abgeleitet.





P-Spec: 2.3

Prozess: 2.3 Name: P16
Input: Wert, Limit Output: - Contr-Out: K3
Body: if Wert > Limit then K3:=

\ Die Namensgebung sollte den Kontrollcharakter verdeutlichen, z.B. Taste_gedrückt, Fehler13, vorwärts, Druck steigend, ...

\ Kontrollflüsse sind i.d.R. binäre Größen, können jedoch auch einen größeren Wertebereich besitzen. Es sind dann aber stets diskrete Wertebreiche!

Je nachdem wie sie in einer Kontrollspezifikation verarbeitet werden, werden Kontrollflüsse bezeichnet als:

Bedingung, wenn sie als Eingang einer Entscheidungstabelle dienen,

Ereignis, falls sie als Eingang eines Zustandsdiagramms dienen,

Aktion, wenn sie als Eingang einer Prozeßaktivierungstabelle dienen.

Die Verarbeitung von Kontrollflüssen in Kontrollspezifikationen wird weiter unten im Detail beschrieben.

5.1.3 Flußdiagramme: Daten- und Kontrollflüsse

Die hierarchische Struktur des **Kontrollmodells** entspricht dem bekannten **Prozeßmodell** von SA. Beide beginnen stets mit dem Kontext-Diagramm. Einige Elemente der Diagramme tauchen immer in beiden Diagrammarten gleich auf, beispielsweise Prozesse und Speicher. Daher bietet es sich an, Daten- und Kontrollflüsse in einem einzigen Diagramm zu zeichnen, einem kombinierten **DFD/CFD.** Dieses bezeichnet man auch als **Flußdiagramm (FD)**. Es beinhaltet dann:

- Prozesse (für DFD und CFD identisch)
- Datenspeicher (für DFD und CFD identisch)
- Datenflüsse (gehören zum DFD)
- Kontrollflüsse (gehören zum CFD)
- Kontrollspezifikationen (oder Barren, gehören zum CFD)

Dies darf solange geschehen, wie die Übersicht nicht darunter leidet.

Als Entscheidungshilfe, wann ein Datenfluss und wann ein Kontrollfluss vorliegt, können folgende Überlegungen dienen:

- Primitive Kontrollflüsse (Signale) sind stets diskret.

- Datenflüsse können stetig und diskret sein.

- Wird die Information verarbeitet, so deutet das auf einen Datenfluss hin. Wird die Information hingegen zur Steuerung verwendet, so deutet das auf einen Kontrollfluss hin.

5.2 Kontrollspezifikationen

Die Kontrollflüsse werden analog zu Datenflüssen an die Stelle des Systems geleitet, wo sie benötigt werden. Zur Unterscheidung, ob ein Kontrollfluss weitergeleitet wird oder in einem Diagramm zur Steuerung benötigt wird, enden letztere auf einem Balken, auch Barren oder Bar genannt. Die Bars bilden die Schnittstelle zwischen den Diagrammen und den zugehörigen Kontrollspezifikationen. Aus Übersichtlichkeitsgründen kann es in einem Diagramm mehrere Bars geben, aber sie gehören alle zu ein und derselben Kontrollspezifikation. Die Kontrollspezifikation dient als Quelle und Ziel für Kontrollflüsse.

Da die Verarbeitung der Kontrollflüsse in den Kontrollspezifikationen erfolgt, legt man zu den CFDs die erforderlichen Kontrollspezifikationen an. In der Regel existiert jedoch nicht zu jedem CFD eine Kontrollspezifikation: es wird nur dann eine angelegt, wenn eine direkte Steuerung der Prozesse des CFDs durch Kontrollflüsse erfolgt. Als sichtbares Zeichen dafür, dass es zu einem CFD eine Kontrollspezifikation gibt, werden die Balken verwendet.

5.2.1 Darstellungsmittel

Um komplexere Steuerungen möglichst klar darstellen zu können, greift man auf Darstellungsmittel der Automatentheorie zurück. Endliche Automaten unterteilen sich in Kombinatorische und Sequentielle Automaten. Bei Kombinatorischen Automaten hängt die Ausgabe einzig von der gegenwärtigen Eingabe ab. Sie können mit Hilfe von Entscheidungstabellen dargestellt werden. Bei Sequentiellen Automaten hängt die Ausgabe von der gegenwärtigen Eingabe und von den vorangegangenen Systemzuständen ab. Sie können mit Hilfe von Zustandsübergangsdiagrammen dargestellt werden.

Neben einer textuellen Beschreibung stehen für Kontrollspezifikationen (**C-Spec**) folgende Darstellungen zur Verfügung:

- Logische Funktionen in Form von
  - Entscheidungstabellen (ET)

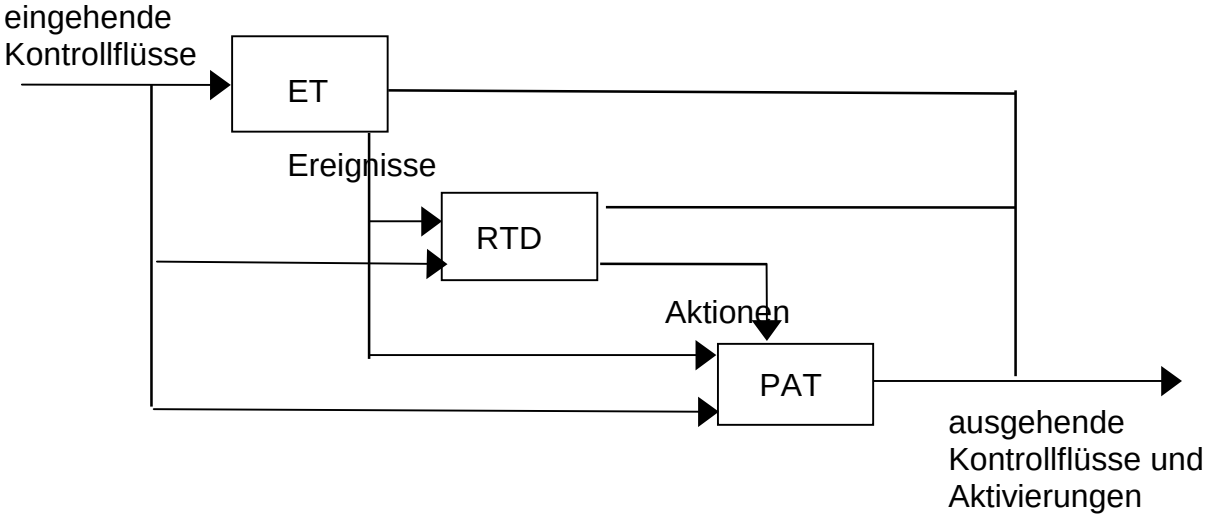
- Endliche Automaten dargestellt durch
  - Zustands (übergangs-) diagramme (RT-Diagramme, RTD)

- Definition der Aktionslogik in Form von
  - Prozeßaktivierungstabellen (PAT)

Auch die Prozeßaktivierungstabellen basieren letztendlich auf kombinatorischen Automaten. Sie sind also mit Entscheidungstabellen verwandt, haben aber eine etwas andere Darstellungsform.

5.2.2 Kombinationsmöglichkeiten

Die drei zuletzt genannten Komponenten können zusammen oder einzeln verwendet werden. Bei einer gemeinsamen Verwendung sind die Komponenten miteinander verknüpft. So können die Ausgängen einer ET Eingänge im RT-Diagramm oder der PAT sein. Ebenso können die Aktionen des RT-Diagramms in der PAT definiert werden. Die möglichen Zusammenhänge zwischen den Komponenten untereinander zeigt die folgende Abbildung:



Das heißt: Die C-Spec-Eingaben werden an Entscheidungstabellen übergeben. Kombinationen von Eingaben wirken als Ereignisse. Die Zustandsübergangsdiagramme erzeugen dann als Folge dieser Ereignisse Aktionen. Diesen Aktionen werden dann in der Prozeßaktivierungstabelle (eigentlich auch eine Entscheidungstabelle) Prozeßaktivierungen und Steuersignale zugeordnet.

5.2.3 Regeln für C-Specs und Kontrollflüsse

Für die C-Specs gelten folgende Regeln:

- Jede Kontrollspezifikation trägt den Namen des zugehörigen (Kontroll-) Flußdiagramms.

- Jeder Prozeß, der in der C-Spec kontrolliert wird, muß im zugehörigen Diagramm erscheinen.

- Alle Kontrollflüsse, die in einem Diagramm auf einen Balken fließen, müssen in der C-Spec als eingehende Flüsse erscheinen.

- Alle Kontrollflüsse, die in einem Diagramm aus einem Balken fließen, müssen in der C-Spec als ausgehende Flüsse erscheinen.

Die beiden letztgenannten Regeln betreffen das sogenannte Horizontale Balancing von Kontrollflüssen. Die Regeln des Vertikalen Balancing von Kontrollflüssen seien hier der Vollständigkeit halber erwähnt, auch wenn sie nicht unmittelbar in Zusammenhang mit C-Specs stehen:

- Alle Kontrollflüsse, die in einen Knoten fließen, müssen in der Verfeinerung entweder in einen Balken oder in einen Knoten fließen.

- Alle Kontrollflüsse, die aus einem Knoten fließen, der verfeinert ist, müssen entweder aus einem Balken oder aus einem Knoten der Verfeinerung fließen.

Weitere spezielle Regeln, die nur einzelne Formen von C-Specs betreffen, werden weiter unten in der Vorstellung der jeweiligen Form genannt.

5.2.4 Textuelle Bescheibung von C-Specs

ToDo

5.3 Entscheidungstabellen

Eine Entscheidungstabelle bestimmt in Abhängigkeit vom Wahrheitsgehalt bestimmter Eingangsbedingungen eine oder mehrere auszuführende Aktionen.

Sie besteht aus den vier Teilen

- **Bedingungen**
- **Eingangsmatrix**
- **Aktionen**
- **Ausgangsmatrix**

Die Spalten der Eingang- und Ausgangsmatrix werden auch als **Regeln** bezeichnet. Jede einzelne Regel beschreibt, unter welchen Bedingungen welche Aktionen ausgeführt werden sollen.

Eine Regel bildet im Bedingungsteil eine Kombination aller Bedingungen. Sie ist dann erfüllt, wenn alle Bedingungen Zustände annehmen, die den der Regel zugeordneten Zuständen entsprechen. Zur Formulierung von Bedingungen gibt es drei mögliche Einträge: J (die Aussage ist wahr), N (die Aussage ist nicht wahr) und - (es ist unerheblich, ob die Aussage wahr oder falsch ist). Ist eine Zelle der erfüllten Regel mit einem Kreuz (X) versehen, dann wird die Aktion derselben Zeile ausgelöst. Sind in einer Spalte mehrere Kreuze gesetzt, dann werden entsprechend mehrere Aktionen ausgeführt.

5.3.1 Einfaches Beispiel

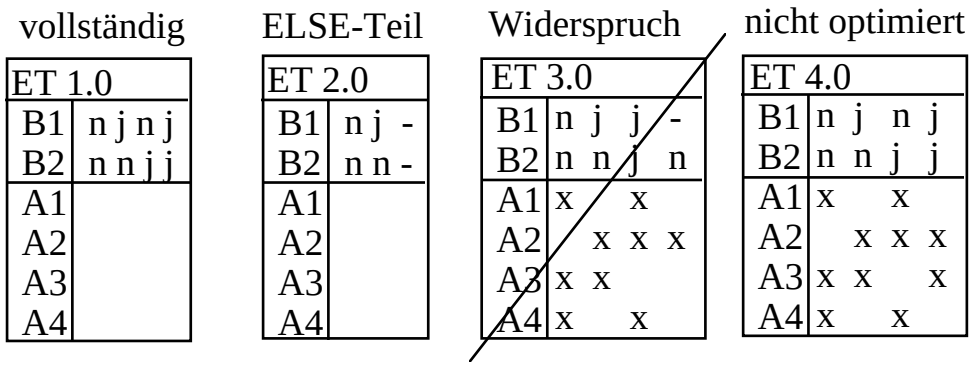
In der folgenden Entscheidungstabelle wird beschrieben, wie die Kunden eines Versandhauses zur Weihnachtszeit betreut werden. Eingangsbedingungen sind der Wert der von einem Kunden während des abgelaufenen Jahres bestellten Waren (BW) und die Kenntnis der Vertriebsmannschaft über die Vorlieben eines Kunden, hier, ob er Alkohol ablehnt oder nicht. Entsprechend dieser Bedingungen werden zur Weihnachtszeit Karten mit oder ohne Präsente verschickt, oder die Wünsche noch zusätzlich in einem persönlichen Anruf übermittelt.

Algorithmus:	Regeln				
Kundenbetreuung	1	2	3	4	5
$0 \leq BW \leq 5000$	J	N	N	N	N
$5000 < BW \leq 20000$	N	J	J	N	N
$20000 < BW$	N	N	N	J	J
kein Alkohol	-	J	N	J	N
Glückwunschkarte	x	x	x	x	x
Weinpräsent			x		x
Buchpräsent		x		x	
persönl. Anruf				x	x

5.3.2 Eigenschaften von Entscheidungstabellen

Es ist zu beachten, daß die Entscheidungstabellen

- vollständig (insbesondere, falls keine ELSE-Spalte vorhanden),
- widerspruchsfrei und
- möglichst optimiert sind.



ET 1.0	
B1	n j n j
B2	n n j j
A1	
A2	
A3	
A4	

ET 2.0	
B1	n j -
B2	n n -
A1	
A2	
A3	
A4	

ET 3.0				
B1	n	j	j	-
B2	n	n	j	n
A1	x		x	
A2		x	x	x
A3	x	x		
A4	x		x	

ET 4.0				
B1	n	j	n	j
B2	n	n	j	j
A1	x		x	
A2		x	x	x
A3	x	x		x
A4	x		x	

Die Tabelle sollte also so aufgebaut sein, dass für jede mögliche Kombination von Zuständen genau eine Regel erfüllt ist. Die SONST-Spalte (ELSE-Teil) ist optional. Sie deckt alle Kombinationen ab, die durch keine der Regeln erfaßt wurden. Sie hat daher im Bedingungsteil nur leere Einträge.

Eine ET heißt **formal vollständig**, wenn alle möglichen Kombinationen der Eingangsbedingungen abgedeckt sind. Formale Vollständigkeit wird in den Beispielen oben in der ET 1.0 und der ET 4.0 erreicht. Formale Vollständigkeit hat Probleme da, wo Bedingungen logisch zusammenhängen, wie im größeren Beispiel der Kundenbetreuung weiter oben. Hier sind 3 Bedingungen über den Bestellwert (BW) formuliert. Wenn man versucht, alle formal möglichen Kombinationen einzutragen, erhält man Widersprüche in diesen Bedingungen. Diese ET kann nicht formal vollständig gemacht werden!

Eine Entscheidungstabelle, die alle tatsächlich möglichen Fälle abdeckt, heißt **logisch vollständig**. Die ET zur Kundenbetreuung ist logisch vollständig.

Durch die Einführung einer ELSE-Spalte wird eine Entscheidungstabelle sowohl logisch als auch formal vollständig. Sie sollte jedoch nur mit Vorsicht gebraucht werden, da eine ET mit ELSE-Spalte nicht mehr auf Vollständigkeit geprüft werden kann.

5.3.3 RT-Entscheidungstabellen

Wie im obigen Beispiel zu sehen, können Entscheidungstabellen vielfältig in der Softwareentwicklung eingesetzt werden. In der konkreten Situation der Beschreibung einer Kontrollspezifikation gelten jedoch noch folgende Randbedingungen:

- Die Bedingungen müssen über die Eingangsflüsse des Barrens formuliert werden.

- Bei der Formulierung der Aktionen können ausfließende Kontrollflüsse, Ereignisse des RT-Diagramms, Aktionen der PAT, oder Prozesse des Diagramms berücksichtigt werden.

Man spricht in diesem Zusammenhang auch von RT-Entscheidungstabellen, weil sie zur Modellierung von Realtime-Eigenschaften eines Systems dienen.

5.3.4 Entscheidungstabellen mit flexiblen Werten

In der bisher vorgestellten Form können die Zellen einer ET nur die festen Werte J,N und - annehmen. Es gibt eine zweite Variante von Entscheidungstabellen, bei der in den Bedingungen nicht unbedingt eine Aussage sondern irgendein Text, wie etwa der Name einer Eingangsgröße des Barrens, steht. Dieser bildet dann zusammen mit dem Zellenwert, der hier einen beliebigen Text annehmen darf, die Bedingung. Die erste Form heißt ET mit festen Zellenwerten, die zweite Form heißt ET mit beliebigen Zellenwerten.

Das obige Beispiel kann als ET mit beliebigen Zellenwerten so formuliert werden:

Algorithmus:	Regeln				
Kundenbetreuung	1	2	3	4	5
BestellWert (BW)	≤ 5000	≤ 20000	≤ 20000	> 20000	> 20000
kein Alkohol	-	J	N	J	N
Glückwunschkarte	x	x	x	x	x
Weinpräsent			x		x
Buchpräsent		x		x	
persönl. Anruf				x	x

5.4 Zustandsübergangsautomaten

Das Verhalten von dynamischen Systemen kann durch Ereignisse geändert werden. Als Folge eines Ereignisses kann ein System in einen anderen Zustand (Betriebsart, Modus, ...) wechseln. Zur Modellierung solcher Systeme dienen dienen die Zustandsübergangsautomaten (Sequentielle Automaten). Es wird ein endlicher Automat beschrieben, der aus verschiedenen Zuständen (States) besteht. Zu jedem Zeitpunkt befindet sich der Automat in genau einem Zustand. Aufgrund von externen Signalen (Events) werden Zustandsübergänge (Transitions) ausgeführt und dabei Aktionen (Actions) ausgelöst. Die Reaktion auf ein Ereignis hängt dabei davon ab, in welchem Zustand sich das System befindet. Sequentielle Automaten haben also ein Gedächtnis.

Ein Zustand resultiert,
- weil das System darauf wartet, dass im externen Umfeld etwas geschieht (Warten auf ein Ereignis), oder
- weil das System darauf wartet, dass eine Tätigkeit durch eine andere abgelöst wird (Beispiel: waschen, mischen, füllen ...).

Beispiele für solche Zustandsänderungen sind:

- Das System geht in eine andere Bearbeitungsphase über, wie z.B. Start, Steigflug, Horizontalflug, Sinkflug, Landung.
- Das System geht in einen anderen Zustand über, wie z.B. Tresortür verschlossen, entriegelt geöffnet.
- Das System geht in einen anderen Betriebsmodus, wie z.B. Testmodus, Normalmodus, Administratormodus.

Reagiert ein System immer gleich, hängt die Aktion, die ausgeführt wird, also immer nur vom Ereignis ab, dann spricht man von einem kombinatorischen Automaten, dieser stellt einen Spezialfall des Sequentiellen Automaten dar (er hat nämlich nur einen Zustand). Kombinatorische Automaten können mit Hilfe einer Entscheidungstabelle beschrieben werden. Hier ist kein Gedächtnis vorhanden, sondern wie schon erwähnt, hängt die Reaktion nicht von einem Zustand ab.

5.4.1 Zustandsübergangsdiagramme

Ein nützliches Hilfsmittel, um Zustandsänderungen zu beschreiben, sind Zustandsübergangsdiagramme. Als Komponenten stehen dabei folgende Elemente zur Verfügung:

- **Zustände** (States)

Ein Zustand wird durch einen rechteckigen Rahmen dargestellt. Im Innern des Rahmens steht der Name des Zustands. Ein Zustand stellt eine Zeitspanne dar, während der das System ein bestimmtes Verhalten zeigt.

- **Übergänge** (Transitions)

Ein Übergang wird durch einen Pfeil zwischen zwei Zuständen dargestellt. Der Pfeil beginnt bei dem Ausgangszustand des Übergangs und zeigt auf den Folgezustand, den das System nach dem Übergang einnimmt.

- **Ereignisse/Aktionen** (Events/Actions)

Ein Event stellt ein Ereignis dar, bei dessen Eintreten ein Zustandsübergang und eine Aktion ausgelöst wird. Die Kombination Ereignis/Aktion wird im Diagramm dem entsprechenden Übergang zugeordnet. Die Kombination kann auch in der folgenden Form notiert werden: Ereignis
Aktion

Es gibt einen ausgezeichneten Zustand, den sogenannten Startzustand. In ihm befindet sich das System zu Beginn. Dieser Zustand ist dadurch zu erkennen, dass ein Pfeil auf ihn zeigt, der keinen Ausgangszustand hat. Dieser Pfeil kann zur Verdeutlichung mit dem Ereignis Start oder Init beschriftet werden.

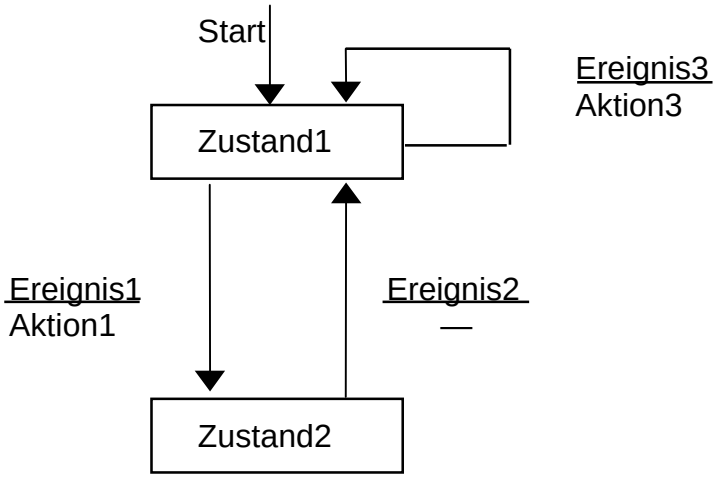
Für die Zustandsübergangsdiagramme gelten folgende Regeln:

- Bei einem Übergang kann die Aktion fehlen, ein Ereignis muss aber immer da sein, das für einen Übergang verantwortlich ist.

- Aus einem Zustand können mehrere Pfeile herausführen. Die Ereignisse, die an diesen Pfeilen stehen, müssen dann disjunkt sein. Ansonsten hätten wir kein deterministisches Verhalten. Es würde ja bedeuten, dass bei ein und demselben Ereignis mehrere Übergänge möglich sind, und es wird nicht spezifiziert, wie sich der Automat verhalten soll.

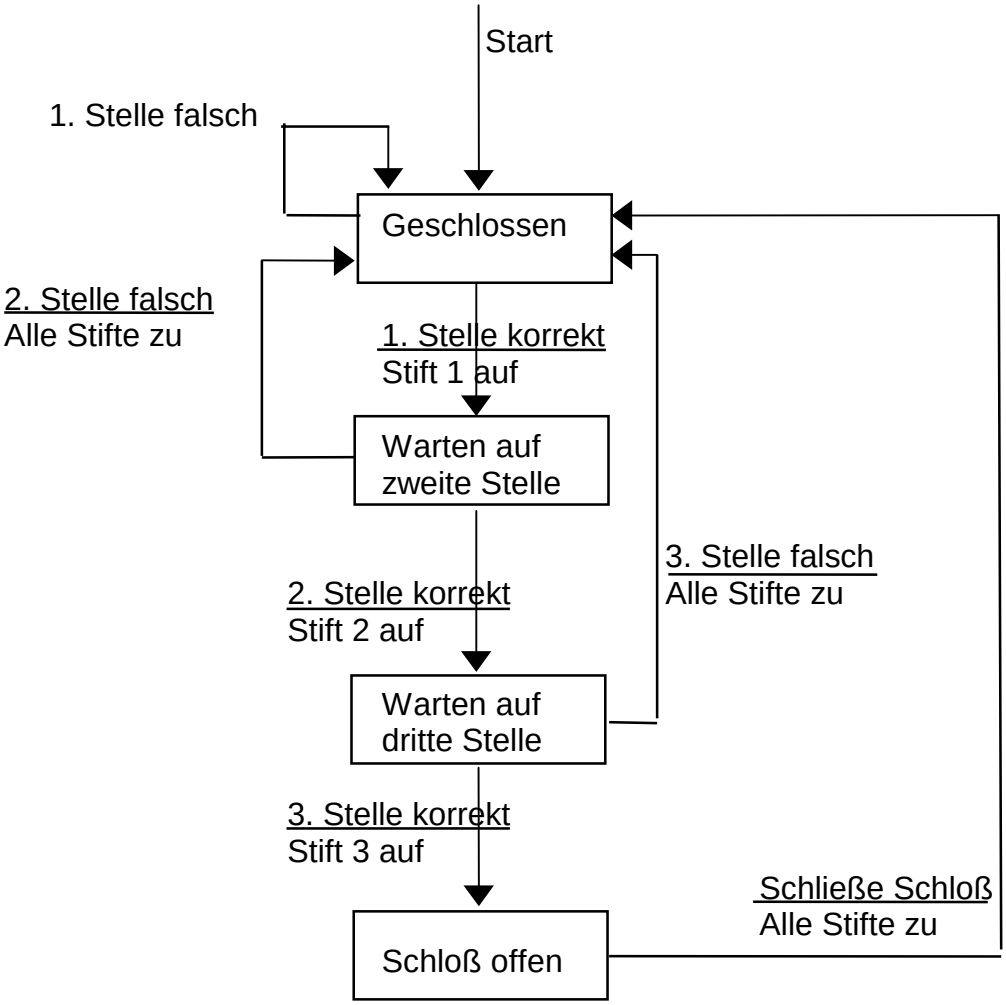
- Übergänge können auch zu dem Zustand führen, von dem sie ausgegangen sind. Dies ist stets dann der Fall, wenn ein Ereignis eine Aktion auslösen, aber keine Zustandsänderung bewirken soll.

Die Möglichkeiten von Zustandsübergangsdiagrammen sind an dem nun folgenden schematischen Beispiel zu sehen.



5.4.2 Einfaches Beispiel

Als Beispiel betrachten wir ein Rotationskombinationsschloß, wie es in den meisten Stahlschränken zu finden ist. Wir beschränken uns hier auf 3 Räder, die eingestellt werden müssen. Die Räder müssen der Reihe nach richtig eingestellt werden. Bei jeder richtigen Einstellung werden die entsprechenden Stifte im Schloß geöffnet, bei einer falschen Einstellung muss wieder ganz von vorne begonnen werden.



5.4.3 Zustandsübergangsmatrix

Neben der Darstellung als Zustandsübergangsdiagramm können die Endlichen Automaten auch durch eine Zustandsübergangsmatrix, also in einer Tabellenform, dargestellt werden. Der Informationsgehalt ist der gleiche. Die grafische Form des Diagramms ist nur leichter verständlich. Aus Übersichtsgründen teilt man die Zustandsübergangsmatrix in zwei Teile auf: den Goto-Teil, der die Zustandsübergänge beschreibt, und den Action-Teil, der die dabei auszuführenden Aktionen enthält.

Der Automat aus dem Beispiel oben könnte damit auch folgendermaßen dargestellt werden:

ToDo

5.4.4 Eigenschaften von RT-Diagrammen

Endliche Automaten oder Zustandsübergangsdiagramme werden in der Informatik an vielen Stellen eingesetzt. Bei dem Anwendungsgebiet hier, nämlich der Modellierung von Kontrolllogik, gelten noch die folgenden zusätzlichen Randbedingungen.

- Ereignisse können in den Barren eingehende Kontrollflüsse sein oder können in der Entscheidungstabelle definiert werden.

- Aktionen können ausgehende Kontrollflüsse betreffen, Prozesse des gleichen Diagramms (de-)aktivieren oder müssen in einer Prozessaktivierungstabelle definiert werden.

Man spricht in diesem Zusammenhang auch von den RT-Diagrammen, weil sie bei der Modellierung von Realtime-Eigenschaften eines Systems eine zentrale Rolle spielen.

5.5 Prozeßaktivierungstabellen

In einer Prozeßaktivierungstabelle werden eingehende Signale in Prozeßaktivierungen und Ausgangssignale umgesetzt. Die Tabelle hat Ähnlichkeiten in der Form und im Inhalt mit Entscheidungstabellen. Der Fokus liegt aber eben auf der Prozeßseite, daher die besondere Form. Die Tabelle ist in drei Spaltengruppen unterteilt:

- Die erste Spaltengruppe besteht nur aus einer Spalte, in der die Eingangssignale aufgelistet sind. Eingangssignale können eingehende Kontrollflüsse, Ausgänge der zugehörigen Entscheidungstabelle oder Aktionen aus dem Zustandsübergangsdiagramm sein.
- In der zweiten Spaltengruppe, der Prozeßgruppe, bilden die zu steuernden Prozesse jeweils eine Spalte. Jede dieser Spalten muß als Überschrift den Namen (oder die Nummer) eines Prozesses (aus dem gleichen Diagramm) tragen.
- In der letzten Gruppe, der Ausgabegruppe, sind die Ausgangssignale des Barrens spaltenweise aufgelistet. Jede dieser Spalten muss als Überschrift den Namen eines Kontrollflusses tragen. Diese Gruppe ist optional.

In einer Prozeßaktivierungstabelle bedeuten auf der Prozeßseite:

1	Prozeß aktivieren,
-1	Prozeß deaktivieren,
0	keine Änderung des Zustandes.

Es können durch ein Eingangssignal mehrere Prozesse aktiviert oder deaktiviert werden. Spielt die Reihenfolge der (De-)Aktivierung eine Rolle, dann wird dies durch entsprechende Zahlenwerte (1,2,3, bzw. -1,-2,-3 usw.) ausgedrückt.

Spielt die Reihenfolge keine Rolle und sind keine Deaktivierungen vorzunehmen, dann wird statt einer 1 manchmal auch ein X in die Tabelle eingetragen. Statt einer 0 ist manchmal auch ein – oder ein leerer Eintrag zu finden. Diese Schreibweise wird auch in der Ausgabegruppe verwendet.

Generell können von einer PAT nur die Prozesse gesteuert werden, die sich mit dem Barren, zu dem die PAT gehört, im selben Diagramm befinden. Prozesse, die von der Kontrollverarbeitung nicht betroffen sind, brauchen natürlich auch nicht in der PAT aufgeführt werden.

5.5.1 Prozeßaktivierung und Prozeßdeaktivierung

Im Rahmen der SA „zündet“ jeder Prozeß sofort, wenn seine Eingaben bereit stehen. Eine Deaktivierung ist im Rahmen der SA nicht möglich. Erhält ein Prozeß sein nächstes Eingabepaket, dann läuft er erneut. Andere Arten von Aktivierungen sowie eine Deaktivierung ist nur im Rahmen der SA/RT möglich. Das grundsätzlich Neue daran ist allerdings nur die Deaktivierung. Für diese Aktionen werden nicht unbedingt die Prozeßaktivierungstabellen benötigt, denn entsprechende Aktionen können auch aus Entscheidungstabellen oder RT-Diagrammen heraus veranlaßt

werden. Für kompliziertere Fälle bietet aber nur die PAT entsprechende Ausdrucksmittel an.

Aktivieren von Prozessen heißt, der Prozeß arbeitet nun tatsächlich datengetrieben, er führt also seine Funktion aus.

- Nur Prozesse, welche nicht über eine C-Spec gesteuert werden, sind **stets aktiv** !
- Prozeßaktivierung erstreckt sich rekursiv über alle Ebenen.
- Ein Prozeß ist nur dann aktiv, wenn alle seine Vorgänger aktiv sind.
- Wird ein Prozeß deaktiviert, so werden rekursiv alle enthaltenen Teilprozesse deaktiviert (auch jene, welche nicht explizit durch eine C-Spec gesteuert werden).

5.5.2 Einfaches Beispiel

Im folgenden Beispiel einer PAT im Zusammenhang mit der Steuerung eines EC-Geldautomaten werden die vier Prozesse Vorgang_Wählen, Karte_Prüfen, Vorgang_Bearbeiten und Karte_Ausgeben durch die Signale Prüfen, Bearbeiten und Auswerfen gesteuert. Bei dem Signal Auswerfen sollen alle Prozesse deaktiviert werden außer Karte_Ausgeben. Nur dieser Prozess wird aktiviert und ist damit der einzige Prozess, der auf Ereignisse reagieren kann. Damit soll ausgedrückt werden, dass der Automat bei der Rückgabe der Karte an den Kunden nicht unterbrochen werden darf.

	Karte_ Prüfen	Vorgang_ Wählen	Vorgang_ Bearbeiten	Karte_ Ausgeben
Prüfen	1	-1	-1	-1
Bearbeiten	-1	1	1	0
Auswerfen	-1	-1	-1	1

In diesem Beispiel sind nur die ersten beiden Spaltengruppen in der PAT vorhanden, da keine Ausgangssignale beeinflußt werden müssen, sondern nur Prozesse gesteuert werden.

5.6 Beispiel

Wir betrachten nun im Folgenden ein komplettes Beispiel, das das Zusammenspiel von SA- und RT-Komponenten demonstriert, aber auch im RT-Teil das Zusammenwirken der unterschiedlichen Darstellungen aufzeigt.

Dem Beispiel „Warenautomat“ liegt folgende Beschreibung zugrunde:

> Der Kunde gibt zunächst seinen Warenwunsch an. Daraufhin wird ihm der Warenpreis angezeigt. Nachdem der Kunde den Betrag bezahlt hat, wird die Ware ausgegeben. Der Automat soll Münzen und Geldscheine in der Landeswährung annehmen. Zuviel gezahltes Geld gibt der Automat als Münzen zurück. Außerdem soll der Automat Falschgeld erkennen. Wünscht der Kunde sein bisher eingeworfenes Geld zurück, so soll der Automat dies nach Drücken der Rückgabetaste ausgeben.

ToDo

5.7 Zusammenfassung

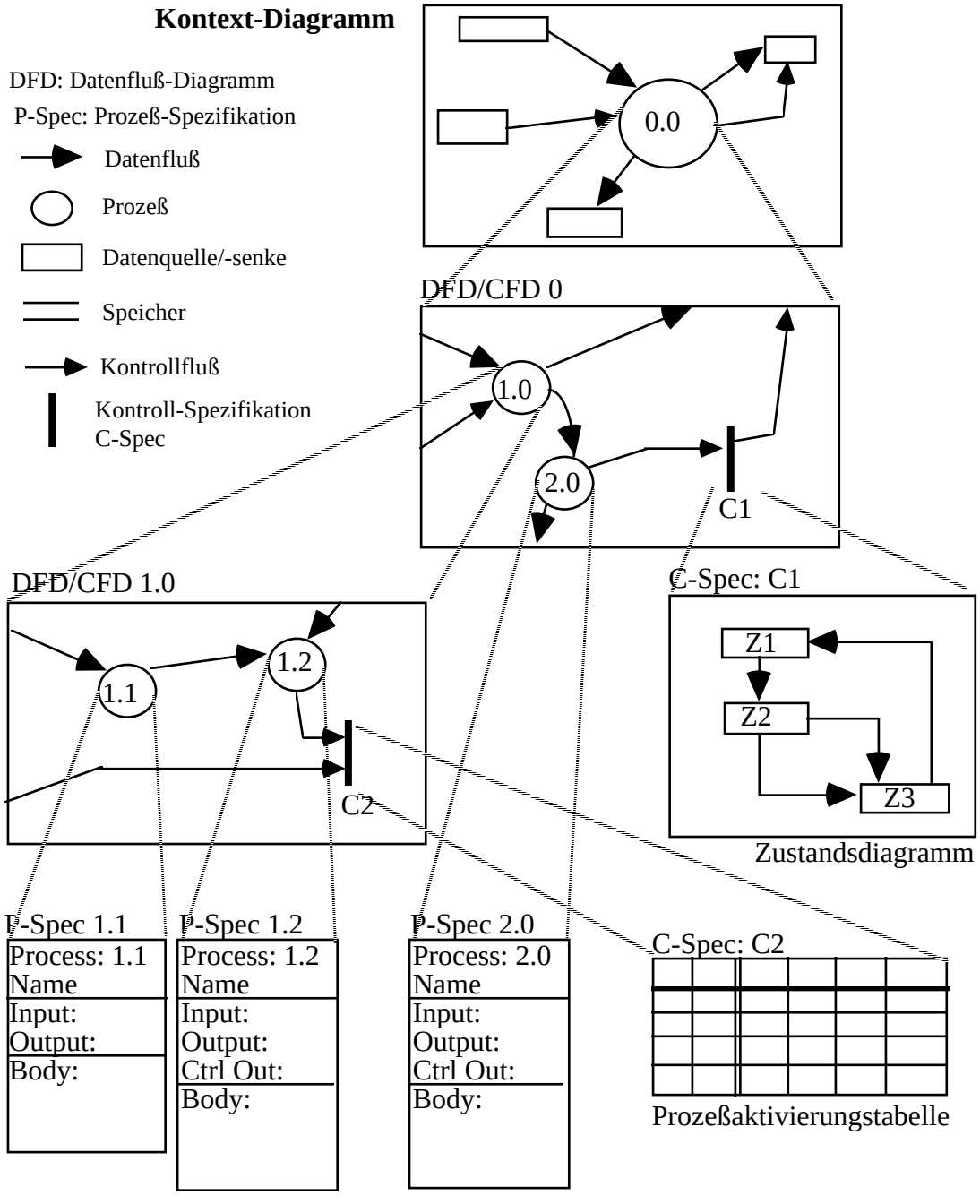
Oftmals zeichnet man Daten- und Kontrollflüsse in ein einziges Diagramm, ein kombiniertes **DFD/CFD.** Dieses bezeichnet man auch als **Flußdiagramm**. Es beinhaltet dann:

- Prozesse (für DFD und CFD identisch)
- Datenspeicher (für DFD und CFD identisch)
- Datenflüsse (gehören zum DFD)
- Kontrollflüsse (gehören zum CFD)
- Kontrollspezifikationen (oder Barren, gehören zum CFD)

Dies wurde bereits in einigen der vorhergehenden Beispiele ausgenutzt. Auf der nächsten Seite befindet sich das Schema für hierarchische DFD/CFD als Zusammenfassung über das ganze Gebiet der SA/RT.

Im Data-Dictionary (DD) werden alle Datenflüsse, Kontrollflüsse, Datenspeicher und Kontrollspeicher definiert und abgelegt.

Schema für hierarchische DFD/CFC im Rahmen von SA/RT:



Data-Dictionary (DD):	**Definition aller Datenflüsse, Kontrollflüsse, Datenspeicher und Kontrollspeicher (auch Teile davon)**

6 Strukturierter Entwurf

Nachdem die Aufgabe (das zu lösende Problem)

- eingehend analysiert ist,
- eine möglichst exakte Aufgabenspezifikation in Form des Pflichtenhefts gemeinsam mit dem Kunden geschaffen wurde und
- sich die Vertragsparter einig sind,

folgt die Phase des Entwurfs (Π **Designphase**).

Für die Entwurfsphase ist es besonders wichtig zu wissen, welche Art der Gesamtlösung angestrebt werden soll:

Traditionelle Methode:
Großer Kostenanteil in eigentliche Programmentwicklung, damit Kosten für Pflege und Wartung (heute 40-70%) möglichst gering gehalten werden können.
Dies sollte der Standard sein!

Quick & Dirty (Q&D)
Entwickler geht davon aus, daß das Programm nur eine kurze Lebensdauer hat, so daß eine ausführliche Vorbereitung und Dokumentation nicht gerechtfertigt ist. Eine Problemanalyse ist kaum vorhanden.
Vorsicht, nichts ist so dauerhaft wie ein gut funktionierendes Provisorium!

Prototyping
I´m really not sure what I want, but I´ll know it when I see it.
Sukzessives Entwickeln verschiedener Versionen des Systems mit Hilfe geeigneter Softwaretools, bis schließlich, unter ständigem Kontakt mit dem Anwender, die fertige Version entsteht. Diese Vorgehensweise ist sehr teuer und wird daher oft nur für grafische User-Interfaces durchgeführt bzw. ist relevant auch nur für diesen Anwendungsbereich.

Wir konzentrieren uns im Folgenden auf die traditionelle Vorgehensweise, für die es eine ganze Reihe von Methoden gibt. Vorgestellt wird die Methode Structured Design (SD), allerdings in einer neueren Version, die Elemente des Modular Design (MD) enthält.

6.1 Übergang von der Analyse zum Entwurf

In der Analyse werden in mehreren Iterationen Anforderungen gesammelt, beschrieben und analysiert, wobei Anforderungen Aussagen über zu erbringende Leistungen sind.

Beim Entwurf wird für die Gesamtfunktionalität des Systems eine Software-Architektur entwickelt, bestehend aus Moduln und ihren Schnittstellen, sowie den Testbedingungen und Testdaten, denen die noch im Rahmen der Implementierung zu realisierende Module später genügen sollen.

6.1.1 Unterschied Analyse - Design

Bei der Analyse geht es darum,

- zu verstehen was der Kunde will,
- die logischen Anforderungen zu erfassen,
- herauszufinden, was die essentiellen Anforderungen sind,
- das Aufgabengebiet zu verstehen,

kurz gesagt, in der Welt der Lösungen zu denken.

Beim Design hingegen gilt es,

- zu verstehen, wie dies realisiert werden kann,
- also physikalische Realisierungsmöglichkeiten zu suchen,
- die Inkarnation der Anforderungen zu beschreiben,
- Strukturen für die Organisation zu suchen und zu finden,

kurz gesagt, in der Welt der Lösungen zu denken.

6.1.2 Von der SA zur SD

Die Elemente der SA (Prozesse, P-Spec, C-Spec) beschreiben lediglich den zeitinvarianten logischen Datenfluß. In der Designphase erfolgt der

Übergang zu später (im Rahmen der Implementierung) konkret

zu realisierenden funktionalen Einheiten.

P-Spec / C-Spec der SA dienen als Vorgabe für den Entwurf, jedoch:

1) Im DFD/CFD existieren z.B. logisch mehrere Prozesse, in denen jeweils Ausgaben (Bildschirm oder Prozeß-I/O) vorgenommen werden. Sie können evtl. geschickt in einem einzigen Modul, welcher alle Ein- / Ausgaben steuert zusammengefaßt werden.

2) In vielen Anwendungen müssen z.B. Daten sortiert werden. Es ist sicherlich nicht sinnvoll nur einen Prozeß zum Sortieren in einem DFD/CFD zu haben. Aber es sollte u.U. nur ein Modul existieren, der die Sortierungen durchführt.

Parallel dazu werden in der Entwurfsphase die Datenstrukturen, die im Data Dictionary beschrieben sind, weiter verfeinert. Die Mittel dazu bietet das Data Dictionary durch die Notation. Darauf wird hier nicht mehr eingegangen.

Der Übergang von der SA zum SD bedeutet, dass das Systemmodell serialisiert werden muss: In der Analyse wurde ein implementationsfreies Modell des Systems erstellt, in dem parallele Prozesse nach dem Datenflussprinzip miteinander kommunizieren. D.h. es werden nur Nachrichten versandt, was mit diesen geschieht, interessiert den Absender nur wenig. Das Designmodell ist an eine bestimmte Implementierungstechnologie gebunden.

Eine solche Technologie kann selbstverständlich auch eine Parallelität ermöglichen, aber hier herrscht nun das Auftragsprinzip, nämlich dass Aufträge erteilt werden und Antworten (Quittungen, Auskünfte, Fehlermeldungen oder sonstige Reaktionen) gefordert werden. Letztendlich muss das Designmodell berücksichtigen, dass auf einer von-Neumann-Architektur, wie sie heute üblich ist, alle Programmbefehle sequentiell abgearbeitet werden, und damit eine feste Reihenfolge der einzelnen Prozesse auf einer Maschine zwingend ist.

6.1.3 Ziele des Entwurfs

Warum braucht man aber überhaupt die Entwurfsphase? Warum kann man nicht gleich das Analysemodell implementieren? Der Unterschied zwischen Analyse und Design besteht, weil

- die verwendete Technologie zu komplex ist,
- die Technologie die Spezifikation einer Vielzahl von technischen Details erfordert,
- die Personen, die die Anwendung verstehen, nicht die gleichen sind, die sie implementieren.

Technologische Randbedingungen, wie Performanz, Zuverlässigkeit, Testbarkeit, Wartbarkeit, Kosten und physikalische Einschränkungen führen zu einem iterativen Vorgehen mit manchmal großen Korrekturen während der Designphase.

Aus diesen Gründen gibt es für einen Design folgende Zielsetzungen:

- Wartbarkeit: neue Komponenten sollen gegen bestehende leicht ausgetauscht werden können.
- Flexibiltät: neue Anforderungen sollen leicht integriert werden können.
- Wiederverwendbarkeit: einzelne Komponenten sollen auch in anderen Software-Systemen eingesetzt werden können.
- Lesbarkeit: die Struktur des Systems soll übersichtlich dargestellt werden.
- Teamarbeit: verschiedene Personen sollen gleichzeitig am System entwickeln können.

Während bei der Analyse die gegenwärtigen Anforderungen beleuchtet werden, muss ein Design auch zukünftige Anforderungen berücksichtigen, um diesen Zielsetzungen gerecht zu werden. Denn das Ziel ist eine modulare Struktur für das System zu schaffen, um den Wartungsaufwand zu reduzieren und die Wiederverwendbarkeit zu steigern. Ein guter Entwurf unterstützt also eine Phase, die erst in einiger Zeit stattfinden wird.

Die während des Entwurfs erstellten Komponenten werden Module genannt. Man lehnt sich bei diesem Begriff bewußt an die Hardware an, wo das modulare Vorgehen schon seit langem erfolgreich praktiziert wird. Damit läßt sich das Ziel des Entwurfs folgendermassen formulieren:

Ziel: **hierarchische Zerlegung des Problems in möglichst unabhängig voneinander realisierbare Module**, welche

- aufeindander aufbauen,
- in ihrer Komplexität überschaubar sind,
- sich gegenseitig nicht beeinflussen (rückwirkungsfrei),
- klar definierte Schnittstellen besitzen.

6.2 Der Modulbegriff

Wie schon oben erwähnt wurde der Begriff Modul bewußt gewählt, um an die Eigenschaften von Moduln in der Hardware-Entwicklung zu erinnern. Dort stellen sie eine klar umgrenzte Aufgabe in einem Gesamtzusammenhang dar. Es sind Bausteine mit einer gewissen Komplexität, deren Interna nicht weiter interessieren. Wichtigstes Kennzeichen ist die Festlegung der Schnittstelle mit dem Ziel der Austauschbarkeit von Moduln mit gleicher Schnittstelle. Die Korrektheit eines Moduls ist ohne Vorkenntnis seiner Verwendung in einem System nachweisbar, in dem man die Realisierung gegen die Schnittstellenspezifikation prüft.

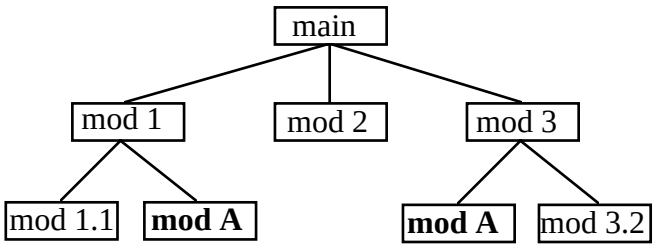
In der geschichtlichen Entwicklung der Entwurfsmethoden wurde dieser Modulbegriff nun auf unterschiedliche Komponenten eines Software-Systems übertragen, was natürlich auch damit zusammenhängt, dass man im Lauf dieser Zeit gelernt hat, immer größere und komplexere Software-Systeme zu entwickeln. Im Folgenden soll dieser Zusammenhang dargestellt werden und der Modulbegriff, so wie wir ihn verwenden wollen, hergeleitet werden.

6.2.1 Structured Design

Structured Design, abgekürzt SD, (von Yourdon/Constantine, Myers) beschäftigt sich mit der Frage:

> Wie kann ein zu entwerfendes Programm geeignet
>
> - in Module zerlegt,
> - die Module untereinander organisiert und
> - ihre Abhängigkeiten in einem Diagramm dargestellt werden.

SD unterstützt die konsequente **Top-Down** Vorgehensweise. Es handelt sich um einen **funktionsorientierten** (d.h. ablauforientierten) Entwurf. Ein Programm besteht dabei aus hierarchisch angeordneten (Programm-) Modulen:



Ein Modul (nicht Prozeß im Sinne der SA) kann hier an mehreren Stellen auftreten und jeweils verschiedene Aufgaben lösen: z.B. **mod A.**

Ein Modul kann (später bei der Implementierung!) realisiert werden als:

- Makro, (d.h. rein textuelle Ersetzung z.B. cpp & #include ..., m4, ...)
- Inline-Code (z.B. für Assembleranweisungen)
- Unterprogramme, Prozeduren, Funktionen in Fortran, Pascal, C,...

Das heißt, es gibt viele Realisierungsmöglichkeiten für einen Modul, je nachdem mit welcher Programmiersprache er später implementiert werden soll.

Die Hierarchie der Module und ihre Abhängigkeiten untereinander werden in Moduldiagrammen und Structure-Charts festgehalten, die Funktionalität der Module in sogenannten M-Specs beschrieben.

6.2.2 Modular Design

Der oben vorgestellte Modulbegriff aus dem SD ist mehr oder weniger identisch mit dem Funktionsbegriff aus den anfangs der 70iger Jahre vorherrschenden Programmiersprachen. Diese dienten vornehmlich zum „Programmieren im Kleinen“.

Basierend auf den Erkenntnissen von Parnas und der Entwicklung im Programmiersprachenbereich (Modula, Ada, OOP) war es notwendig den Begriff des Moduls zu erweitern. Nicht mehr nur einzelne Funktionen stehen im Mittelpunkt sondern die Zusammenfassung von Funktionen und/oder Daten zu größeren Einheiten. Man sprach vom „Programmieren im Großen“ und die dafür entwickelte Methode erhielt den Namen Modular Design (MD). Der Begriff des Moduls wurde dabei nun für diese größeren Einheiten verwendet.

Zentraler Gedanke ist dabei die Kapselung von Daten und Funktionen als Einheit und andererseits die Definition der Schnittstellen zwischen diesen Einheiten, um so zu mehr Prüfbarkeit zwischen den Einheiten zu kommen: Über eine Exportschnittstelle stellt ein Modul Ressourcen für andere Moduln zur Verfügung. Alle anderen Interna sind verborgen. Zur Implementierung eines Moduls kann man andere Moduln benutzen, die man in einer Importschnittstelle auflistet. Damit hat man folgendes erreicht:

- Ein Modul ist ersetzbar durch einen Modul mit gleicher Exportschnittstelle.
- Die Korrektheit eines Moduls ist ohne Kenntnis seiner Verwendung nachweisbar, in dem man die Realisierung gegen seine Export- und Importschnittstelle prüft.
- Der Modulentwickler hat die Kontrolle darüber, was von seinem Modul benutzt werden kann, und was verborgen bleibt.

Damit der Modulbegriff nicht überladen wird, werden die Bestandteile eines Moduls, also die Funktionen, bzw. Daten nicht mehr als Module wie im SD bezeichnet, sondern man spricht nun von Operationen (um allgemein zu bleiben und sich nicht an einen bestimmten Funktionsbegriff einer Programmiersprache anzulehnen). In dieser Terminologie fassen Module Operationen und Daten zu einer Einheit zusammen.

Bei der Realisierung eines Moduls mit einer konkreten Programmiersprache gibt es zwei Möglichkeiten:

- In der Programmiersprache gibt es Module als sprachliche Einheiten, wie Module in Modula oder Package in Ada. Dann wird ein Modul aus dem Design auf einen solchen Modul der Programmiersprache abgebildet.
- In der Programmiersprache gibt es diesen Modulbegriff nicht, sondern nur Funktionen, wie in den meisten älteren Programmiersprachen beispielsweise C oder Pascal. In diesem Fall wird ein Modul aus dem Design abgebildet auf eine übersetzbare Einheit in der Programmiersprache, in aller Regel also auf eine Datei.

Bleiben wir beim Fall der Programmiersprache C, dann werden also die Operationen aus dem MD abgebildet auf Funktionen in C und die Module auf eine Datei, die ja in C mehrere Funktionen und/oder Datendefinitionen enthalten kann.

6.2.3 Hierarchiebildung mit Paketen

In neueren Ausprägungen des MD hat man noch den Begriff des Pakets (Package) eingeführt, aus der Erkenntnis heraus, dass man noch größere Einheiten bilden muss, um die Komplexität neuerer Software-Systeme in den Griff zu bekommen, und um insbesondere eine Hierarchiebildung zu ermöglichen. Denn nach den oben definierten Begriffen enthält ein Modul nur Operationen und kann nicht weitere Module enthalten. Operationen lassen sich sogar gar nicht mehr weiter zerlegen.

Ein Paket kann Module enthalten, aber auch weitere Pakete. Damit ist eine Hierarchiebildung gewährleistet. Dieser Paketbegriff lehnt sich an Java an. In der Realisierung wird ein Paket zu einem Dateiordner (Verzeichnis). Statt des Begriffs Paket wird häufig auch der Begriff eines Subsystems verwendet.

Pakete werden im Folgenden nicht weiter behandelt. Sie stellen wie gesagt eine Erweiterung des MD dar, dessen Grundlagen wir zuerst erarbeiten müssen.

6.2.4 Dynamische Strukturen mit Tasks

Wir benutzen Operationen, Moduln und Pakete um die statischen Strukturen eines Systems auf verschiedenen Abstraktionsebenen zu modellieren. In weniger komplexen Systemen ist das auch ausreichend, um ein System, seinen Aufbau und seine Organisation zu verstehen, denn seine dynamisch Struktur ist einfach und besteht aus einer Ablaufstruktur, dem sogenannten Hauptprogramm.

Heutige System, beispielsweise im Client-Server-Bereich, bestehen aus mehreren Ablaufstrukturen, die teils unabhängig voneinander sind, teils aber auch miteinander kommunizieren, um Dienste des jeweils anderen in Anspruch zu nehmen. Das dynamische Verhalten solcher Systeme kann mit den Mitteln des Strukturierten Designs oder des Modular Designs nicht beschrieben werden.

Hierfür wird eine weitere Designkomponente benötigt, die als Task bezeichnet wird. In neueren Programmiersprachen lassen sich diese direkt realisieren, beispielsweise

durch Tasks in Ada oder Threads in Java. Ansonsten besteht die Möglichkeit, eine Task auf ein ablauffähiges (Haupt-)Programm abzubilden, was man beispielsweise in C tun würde.

Wir werden uns im Folgenden nur mit den statischen Strukturen befassen und Tasks nicht weiter betrachten.

6.2.5 Objekt-Orientierter Entwurf

Der Objekt-Orientierte Entwurf (OOD) entstand Mitte der 80iger Jahre und basiert auf Ideen von G. Booch, B. Meyer und der Smalltalk-Schule. Er betont neben der Kapselung und der Definition von Schnittstellen den Aufbau von wiederverwendbaren Designkomponenten. Wiederverwendbare Klassen werden durch Anwendung des Vererbungsprinzips erstellt.

Die Konzepte werden durch objekt-orientierte Programmiersprachen wie Smalltalk, Eiffel, C++ oder Java unterstützt.

Neben dem „Programming in the Large“ nimmt die Bedeutung des „Programming in the Many“ immer mehr zu. Damit ist gemeint, dass man nicht einen Entwurf für ein System macht, sondern dass in vielen Systemen ähnliche Probleme zu lösen sind. Man versucht für alle diese ähnlichen Probleme einen Entwurf zu machen und über die objekt-orientierten Mechanismen wie Vererbung den Entwurf an konkrete Probleme anzupassen. Es enstehen Entwurfsmuster (design patterns), die aus einem Gerüst von Klassen bestehen und auf viele Anwendungen übertragbar sind.

Objekt-orientierter Entwurf ist Gegenstand einer anderen Vorlesung und wird hier nicht weiter vertieft.

6.2.6 Zusammenfassung

Wir können den Begriff Modul ganz allgemein verwenden, wenn wir eine Komponente bezeichnen, die während des Entwurfs erstellt wird. Komponente oder Baustein sind Synonyme für diesen Modulbegriff.

In der Methode, die im Folgenden vorgestellt werden soll, wird der Begriff Modul allerdings auch noch verwendet für eine ganz bestimmte Art von Komponente. Als Entwurfskomponenten stehen nämlich zur Verfügung:

- Operationen,
- Moduln,
- Packages,
- Tasks.

In diesem Zusammenhang bezeichnet der Begriff Modul die Zusammenfassung von Operationen und Daten zu einem neuen Baustein.

6.3 Strukturen eines Entwurfs

Wenn man einen Entwurf aufbaut, kann man dies grundsätzlich danach ausrichten, wie die Komponenten zusammenspielen sollen. Es enstehen zwei unterschiedliche Entwurfsstrukturen:

- eine Hierarchie mit zentraler Steuerung und Kontrolle oder
- ein Netzwerk mit dezentraler Steuerung (verteiltes System).

Bei Netzwerken sind die Komponenten datengetriggert, gleichwertige Komponenten kommunizieren miteinander. Jede Komponente kann wie ein eigenes System betrachtet werden. Bei einer Hierarchie werden die Komponenten aus einer baumartigen Struktur gesteuert. Die Komponenten können nur über die hierarchische Struktur miteinander kommunizieren. Die Komponenten werden vom nächst höheren Level gesteuert, sie hängen von einander ab.

Software-Systeme müssen nicht eindeutig netzwerkartig oder hierarchisch strukturiert sein. Netzwerkartige Strukturen findet man beispielsweise in Client-Server-Architekturen oder immer dann, wenn ein System auch physikalisch auf mehreren Rechnern verteilt ist. In diesem Falle kann es dann so sein, dass die einzelnen Komponenten eines Netzwerks, beispielsweise der Client und der Server, in sich wieder hierarchisch aufgebaut sind. D.h. also, dass Netzwerk und Hierarchie zusammen vorkommen können.

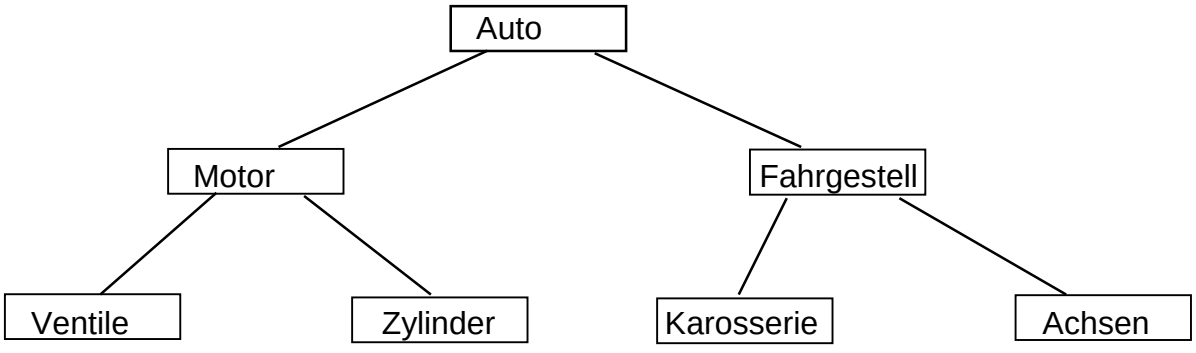
Wir werden uns im Folgenden hauptsächlich mit Hierarchien beschäftigen. Im wesentlichen geht es beim Design um zwei Hierarchien:

Zusammengesetzt-Aus Hierarchie (Composed-Of)
Benutzt Hierarchie (Use)

Netzwerke lassen wir unberücksichtigt, weil sie wie oben angedeutet meist dazu verwendet werden, größere Systeme auch physikalisch zu verteilen und wir uns zuerst einmal mit den grundlegenden Methoden für kleinere Systeme vertraut machen wollen.

6.3.1 Composed-Of Hierarchie

In einer Composed-Of Hierarchie wird dargestellt, aus welchen Einzelteilen sich ein größeres Ganzes zusammensetzt. Beispiel:



Eine Composed-Of Hierarchie wird aus zwei wesentlichen Gründen eingeführt:

- Abstraktion, um die Komplexität zu verringern und Komponenten (egal ob Netzwerk oder Hierarchie) auf einer höheren Abstraktionsebene zu diskutieren.
- Sichtbarkeit (Scoping), um Seiteneffekte bei einer Änderung zu reduzieren und um die Anzahl der erreichbaren Objekte zu begrenzen.

Durch Abstraktion wird es möglich einen Systembereich als geschlossene Einheit zu betrachten, anstatt vieler einzelner Komponenten, beispielsweise ein Betriebssystem statt der einzelnen Betriebssystemfunktionen. Abstraktion ist Vereinfachung der komplexen Realität durch Reduzierung auf das Wesentliche. Grobe Verallgemeinerung statt Blick auf das Besondere und Einzelne stehen im Vordergrund.

Abstrahiert werden können Aktionen (Funktionale Abstraktion) und Daten (Datenabstraktion). Funktionale Abstraktion bedeutet, dass eine Funktion genutzt werden kann unter Angabe ihres Namens und Beachtung ihrer Schnittstelle (Ein- und Ausgabeparameter) aber ohne Kenntnis ihrer Implementierung. Zur funktionalen Abstraktion dienen im Entwurf die Operationen. Funktionale Abstraktion bedeutet eine Lokalität der Kontrolle über Aktionen. Datenabstraktion führt zu einer Lokalität der Daten und der mit ihnen verbundenen Operationen. Bei Datenabstraktion ist der Zugriff auf Daten nur über die definierten Zugriffsoperationen möglich. Man spricht in dem Zusammenhang auch von abstrakten Datentypen. Ein abstrakter Datentyp ist ausschließlich über die auf ihn möglichen Operationen definiert. Er besitzt somit die alleinige Kontrolle über die Datenkonsistenz und über die korrekte Nutzung der Daten.

In dem man in einer Hierarchie Sichtbarkeiten einschränkt (Scoping) erreicht man eines der wichtigsten Ziele, Information Hiding. Durch die Definition von Scopes kann man interne Objekte vor unbefugtem Zugriff verstecken. Durch die Verwendung von Scopes wird die Sichtbarkeit und damit die Benutzung von Objekten eingeschränkt. Je kleiner der Scope, um so weniger Seiteneffekte sind zu erwarten.

Als Scope eines Objektes bezeichnet man den Bereich, in dem ein Objekt gültig ist. Innerhalb dieses Scopes ist das Objekt bekannt für alle anderen Objekte im gleichen Scope und kann von diesen genutzt werden. Praktisch bedeutet dies, dass man innerhalb eines Scopes nicht eine Komponente ändern kann, ohne dass nicht zumindest noch eine andere ebenfalls modifiziert werden muss, oder dies zumindest geprüft werden muss.

Für die Scopes iinerhalb eines Designs gelten folgende Regeln:

- Alles was innerhalb einer Operation definiert ist, ist nur dort sichtbar. Andere Objekte können interne Objekte nicht sehen und nicht benutzen.

- Alles was innerhalb eines Moduls definiert ist, kann von den Modulkomponenten ohne Einschränkung benutzt werden. So können die Operationen eines Moduls sich gegenseitig aufrufen oder die im Modul definierten Daten benutzen.

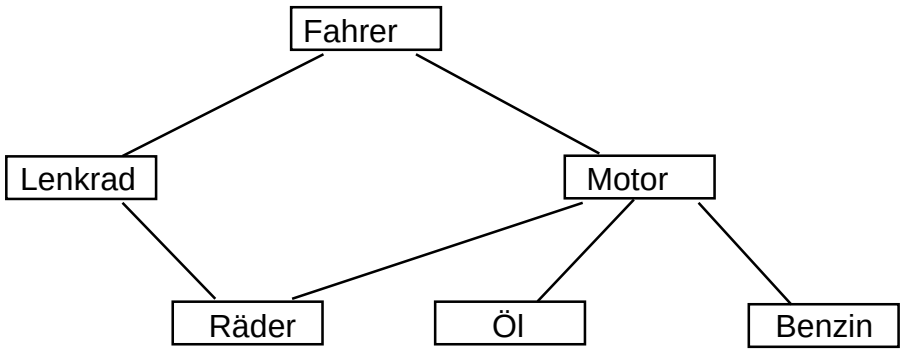
Um Composed-Of Hierarchien darzustellen, gibt es im Design folgende Möglichkeiten:

- Packages bestehen aus (composed-of) Packages und Moduln.
- Moduln bestehen aus Operationen und Daten.
- Operationen bestehen aus Anweisungen und Daten.

Die Composed-Of Hierarchie wird meist durch einen Baum dargestellt, wie bereits oben im einleitenden Beispiel zu sehen war. Man spricht vom sogenannten Hierachie-Diagramm oder vom Modulbaum (Projektbaum).

6.3.2 Use Hierarchie

Eine andere Art von Hierarchie stellen die Use Hierarchien dar. Hier sieht man, wie die einzelnen Komponenten sich gegenseitig benutzen. Beispiel:



Use Hierarchien sind nötig, um die organisatorischen Strukturen in einem Software-System identifizieren. Sie zeigen deutlich, wer der Chef ist und wer die Arbeit durchführt. Sie zeigen insbesondere auf, wo Schnittstellen zwischen den Designkomponenten benötigt werden.

Um Use Hierarchien darzustellen, gibt es im Design die folgenden Möglichkeiten:

- Module benutzen andere Module. Dies wird in sogenannten Moduldiagrammen visualisiert.
- Operationen benutzen andere Operationen und globale Daten. Dies wird in sogenannten Operationsdiagrammen (Structure-Charts) visualisiert.

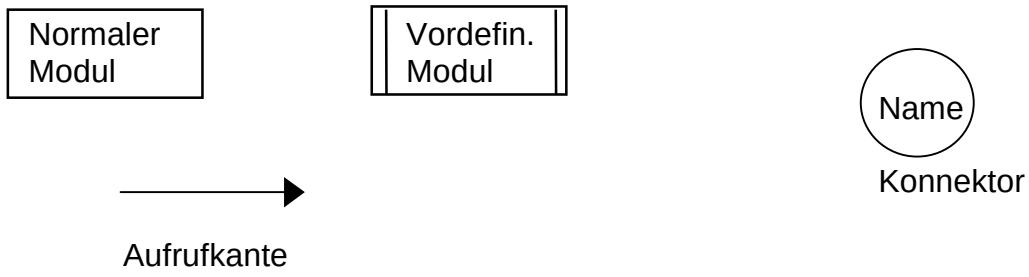
Wie bereits im obigen Beispiel zu sehen, müssen Use Hierarchien nicht mehr baumartig strukturiert sein.

6.4 Moduldiagramme

Ein Moduldiagramm beschreibt die statische Aufrufstruktur (Benutzt-Beziehung) der Module eines Programms. Da Module selbst keine Komponenten darstellen, die aufrufen, sondern nur aus Operationen und Daten bestehen, werden die Aufrufbeziehungen ihrer Operationen, bzw. die Benutztbeziehung ihrer Daten dargestellt. Die Operationen und Daten sind allerdings in diesem Diagramm nicht vertreten, sondern, wie der Name sagt, handelt es sich um ein Diagramm, in dem Module die Hauptrolle spielen. In einem Moduldiagramm wird die Abhängigkeit der Module untereinander dargestellt. Die Beziehung zwischen Operationen und Daten werden in einem sogenannten Operationsdiagramm spezifiziert, das weiter unten erläutert wird.

6.4.1 Aufbau eines Moduldiagramms

Als Elemente eines Moduldiagramms stehen folgende Symbole zur Verfügung:



Module sind die Knoten des Diagramms. Sie werden als Rechtecke dargestellt. Es sind normale und vordefinierte Module zu unterscheiden. Ein Modul wird als normaler Modul dargestellt, wenn er in Rahmen des Projektes zu erstellen ist und nicht schon existiert. Normale Module bestehen aus normalen Operationen und/oder normalen Daten (siehe unten). Ein normaler Modul wird als Rechteck dargestellt, in dessen Inneren der Name des Moduls steht.

Vordefinierte Module (Bibliotheksmodule) bestehen aus vordefinierten Operationen und/oder vordefinierten Daten. Vordefinierte Module werden durch ein Rechteck mit zwei zusätzlichen senkrechten Linien dargestellt (gerahmtes Rechteck), in dessen Inneren der Name des Moduls steht. Dieses Symbol wird verwendet, wenn ein Modul bereits fertig existiert, sei es dass er eingekauft wurde oder im Rahmen eines anderen Projektes entwickelt wurde. Vordefinierte Module brauchen daher nicht mehr ausführlich spezifiziert zu werden.

- Zwei Module A, B werden durch eine gerichtete Kante (Pfeil) von A nach B verbunden, falls eine Operation des Moduls A einen Aufruf einer Operation des Moduls B enthält, oder auf Daten, die zum Modul B gehören, zugreift.
- Die Pfeile tragen keinen Namen. Der Pfeil beginnt also stets beim „aufrufenden“ Modul, die Spitze zeigt auf den „aufgerufenen Modul“.
- Ein Pfeil von Modul A zu Modul B drückt damit aus, dass Modul A zu seiner Realisierung Modul B benötigt, dass also Modul A von Modul B abhängt.

Zyklische Abhängigkeiten, also Modul A hängt vom Modul B und umgekehrt, könnten theoretisch existieren, die meisten Methoden leiten jedoch dazu an, solche Abhängigkeiten zu vermeiden. Von zyklischen Abhängigkeiten spricht man auch dann, wenn die gegenseitige Abhängigkeit zwischen zwei Modulen über mehrere Stufen, also über mehrere Zwischenmoduln, gegeben ist.

6.4.2 Beispiel eines Moduldiagramms

ToDo

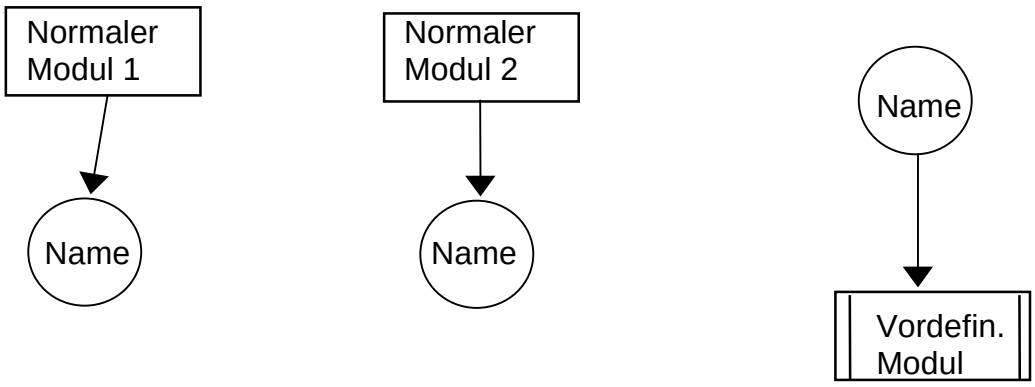
6.4.3 Konnektoren

Ein Konnektor dient als graphisches Hilfsmittel, um die Fortsetzung einer Modulverbindung anzuzeigen. Er erlaubt eine bessere optische Darstellung eines Diagramms und wird hauptsächlich dann eingesetzt, wenn sich mehrere Linien kreuzen. Konnektoren werden durch einen Kreis dargestellt, in dessen Inneren sich der Name des Konnektors befindet.

Man unterscheidet Start- und Zielkonnektor, je nach dem wohin der Pfeil zeigt. Startkonnektoren zeigen den Beginn einer Aufrufstruktur an, der Pfeil geht vom Konnektor auf genau einen „aufgerufenen“ Modul. Zielkonnektoren zeigen eine Unterbrechung einer Aufrufstruktur an, die Fortsetzung stellt der Startkonnektor mit dem gleichen Namen dar. Ein Zielkonnektor hängt an genau einem Modul, d.h. ein Pfeil zeigt von einem Modul auf einen Zielkonnektor.

Zu einem Startkonnektor kann es mehrere Zielkonnektoren geben, die alle den gleichen Namen tragen. Mit anderen Worten: an verschiedenen Stellen des Moduldiagramms kann auf die gleiche Fortsetzung verwiesen werden. Jeder Startkonnektor ist dagegen eindeutig. Es kann nur eine Stelle geben, an der dieser Konnektor als Ausgangspunkt einer Aufrufstruktur verwendet wird.

Das folgende Beispiel zeigt die Verwendung von Konnektoren in einem Moduldiagramm.



Konnektoren können auf die gleiche Art und Weise in Operationsdiagrammen verwendet werden, um diese optisch zu verbessern.

6.4.4 Beschreibung von Moduln

Jeder in einem Moduldiagramm verwendete Modul muß auch beschrieben werden. Man unterscheidet drei Teile in einer solchen Beschreibung:

- **Aufgabenbeschreibung**

Welche Funktion erfüllen die in dem Modul zusammengefaßten Operationen und Daten. Warum wurden Sie zusammengefaßt?

+ **Schnittstellenbeschreibung**

  In der Schnittstellenbeschreibung sollte dargelegt werden, welche der Einheiten (Ressourcen) durch andere Moduln benutzt werden können (Exportschnittstelle) und welche Ressourcen anderer Moduln wie und warum benutzt werden (Importschnittstelle).

+ **Realisierungsbeschreibung**

  Aus welchen Komponenten besteht überhaupt der Modul.

Die Aufgabenbeschreibung erfolgt in einer verbalen textuellen Form. Für die Schnittstellenbeschreibung gibt es eine ganze Reihe formaler Ansätze, die auch eine Überprüfung auf konsistente Einhaltung ermöglichen. Sie sind allerdings nicht in allen Werkzeugen vorzufinden. So ist etwa im Innovator eine Spezifikation der Schnittstellen von Moduln gar nicht vorgesehen. Sie können höchstens textuell zusammen mit der Aufgabenbeschreibung abgefaßt werden, wodurch eine Überprüfung dieser Information nicht möglich ist.

Die Realisierungsbeschreibung stellt die Composed-Of Hierarchie dar und wird in Form eines Baumes visualisiert. Grundlage dafür ist eine Zuordnung von Operationen und Daten zu Moduln.

6.4.5 Eigenschaften des Moduldiagramms

Ein Moduldiagramm enthält die statische Aufrufstruktur des Systems auf Modulebene. Es enthält keine Abbildung des tatsächlichen dynamischen Programmablaufs. Aufrufbeziehungen werden also dargestellt, unabhängig davon, ob die Aufrufe in einer Alternative stehen, die vielleicht dynamisch nie während des Programmlaufs erreicht wird, oder ob sie in einer Schleife stehen, die während des Programmlaufs 1000 und mehr Mal ausgeführt wird.

Das Diagramm läßt sich aus den Operationsdiagrammen herleiten, jedoch erhält man eine wirksame Konsistenzprüfung nur dann, wenn man beide Diagramme getrennt entwickelt.

Gerade für die einfachen Beispiele der Vorlesung sehen die Moduldiagramme einfach aus, sie haben jedoch schon ihre Berechtigung und ihren Nutzen in der Programmentwicklung: beispielsweise können Includes und Externvereinbarungen für den Code daraus abgeleitet werden.

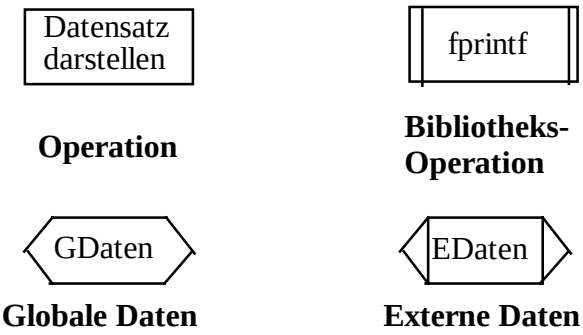
In einem Moduldiagramm ist nicht die Schnittstelle der Aufrufe zu sehen. Dies kann auch auf der Ebene gar nicht geschehen, da ein Modul ja eine Zusammenfassung von mehreren Operationen darstellt, und auch wenn sich viele Operationen aus zwei Moduln aufrufen, dies nur mit einem Pfeil dargestellt wird. Die Aufrufschnittstelle kann also nur auf der Basis von Operationen dargestellt werden, dazu dienen die Operationsdiagramme.

6.5 Operationsdiagramme

Eine Darstellung mit detaillierterem Informationsgehalt bieten die Operationsdiagramme, oder auch Structure-Charts genannt. Man unterscheidet die Art der Operationen und kann präzise ihre Verknüpfungen, d.h. ihre Schnittstelle beim Aufruf beschreiben.

6.5.1 Elemente eines Operationsdiagramms

Es gibt vier Arten von Knoten in einem Operationsdiagramm::



Diese Knoten werden mit Pfeilen (gerichtete Kanten) verbunden, die den Aufruf einer Operation, bzw. die Benutzung von Daten symbolisieren. In diesen Diagrammen können nun darüberhinaus auch die Schnittstelle der Aufrufe (Parameter) spezifiziert werden.

SD-Operationen repräsentieren Funktionen oder Prozeduren einer Programmiersprache. Es sind normale und vordefinierte Operationen zu unterscheiden.

- Eine Operation wird als normale Operation dargestellt, wenn sie in diesem Projekt zu erstellen ist und nicht schon vordefiniert ist. Eine oder mehrere Operationen können einem Modul zugeordnet werden, das in einem Moduldiagramm bearbeitet wird. Operationen werden durch ein Rechteck dargestellt, in dessen Inneren der Name der Operation steht. Zu jeder Operation kann der strukturierte Quellcode (Pseudocode oder Struktogramm) einer Funktion eingegeben werden.

- Vordefinierte Operationen repräsentieren Funktionen oder Prozeduren, die bereits vorhanden sind, z.B. in bestehenden Libraries, und lediglich in das neue Projekt eingebunden werden müssen. Man spricht deshalb manchmal auch von Bibliotheksoperationen. Vordefinierte Operationen werden durch ein Rechteck mit zwei zusätzlichen parallelen Linien dargestellt. Im Inneren des Rechtecks steht der Name der Operation. Vordefinierte Operationen können vordefinierten Moduln zugeordnet werden. Einer vordefinierten Operation kann kein strukturierter Quellcode zugeordnet werden.

SD-Daten repräsentieren Datenbereiche einer Programmiersprache wie z.B. Variablendeklarationen. Sie können nur am Ende einer Aufrufkante liegen. Es sind normale und vordefinierte Datenbereiche zu unetrscheiden.

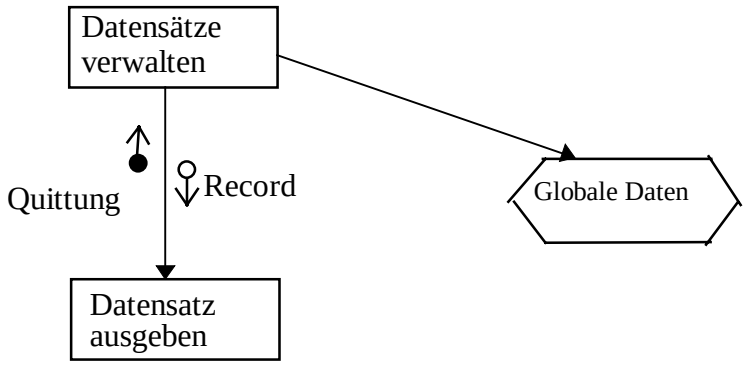
- Ein Datenbereich wird als normale Daten dargestellt, wenn er in diesem Projekt zu erstellen ist und nicht schon vordefiniert ist. Ein oder mehrere Datenbereiche können einem Modul zugeordnet werden unabhängig davon, ob diesem Modul auch Operationen zugeordnet sind. Normale Daten werden durch ein Sechseck dargestellt, in dessen Inneren der Name des Datenbereichs steht. Zu jedem Datenbereich kann Quellcode eingegeben werden, der z.B. aus Variablen- und Typdeklarationen bestehen kann, jedoch nicht aus Funktionen.

- Vordefinierte Daten repräsentieren bereits vorhandene Datenbereiche, z.B. in bestehenden Libraries, die lediglich in das neue Projekt eingebunden werden müssen. Vordefinierte Daten werden durch ein Sechseck mit zwei zusätzlichen parallelen Linien dargestellt, in dessen Inneren der Name des Datenbereichs steht. Vordefinierten Daten kann kein Quellcode zugeordnet werden.

Weiterhin können Konnektoren benutzt werden. Sie werden gleich gezeichnet wie in Moduldiagrammen und unterliegen den gleichen Regeln, wie sie bereits oben beschrieben wurden.

6.5.2 Verknüpfungen im Operationsdiagramm

Operationen werden durch Pfeile verknüpft, welche von der übergeordneten/ aufrufenden Operation zur untergeordneten/aufgerufenen Operation gerichtet sind:



Eine Aufrufkante trägt keinen Namen, ihr können aber wie im Beispiel zu sehen, die Parameter zugeordnet werden, die beim Aufruf notwendig sind. Parameter werden im nächsten Abschnitt detailliert erläutert.

Datenknoten können nur am Endpunkt einer Kante liegen, nicht am Anfang. Eine Kante zu einem Datenknoten bedeutet, dass die dadurch repräsentierten Daten durch die Operation benutzt werden, sei es lesend oder schreibend.

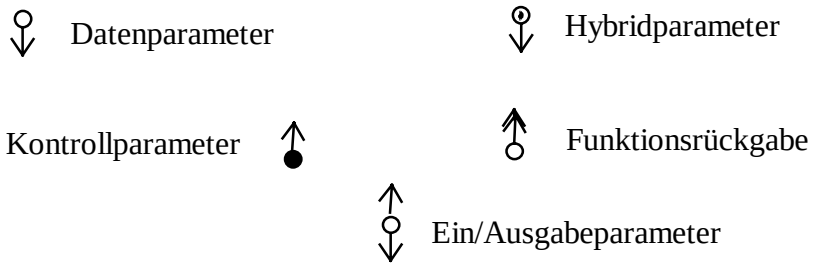
6.5.3 Aufrufschnittstellen

Die Schnittstellen beim jeweiligen Aufruf können (müssen) mitangegeben werden. Dabei werden sowohl die Namen der Parameter und deren Richtung festgelegt. Parameter werden durch kleine Pfeile mit einem kleinen Kreis am Anfang dargestellt, neben der Aufrufkante positioniert und mit dem Namen des Parameters beschriftet. Der Pfeil repräsentiert immer einen aktuellen und nie einen formalen Parameter.

Zeigt der Pfeil zur aufgerufenen Operation, handelt es sich um einen Eingabeparameter, zeigt er zur aufrufenden Operation, so handelt es sich um einen Ausgabeparameter. Parameter, die sowohl Eingabe- als auch Ausgabeparameter sind, haben zwei kleine Pfeile, die in beide Richtungen zeigen.

Datenparameter beschreiben normale Daten, die verarbeitet werden sollen. Sie haben einen leeren Kreis am Kantenanfang.

Kontrollparameter beschreiben besondere Steuerelemente, wie z.B. Schalterstellungen, Signale, Ausnahme- und Fehlersituationen (Flag, EOF, ALARM,...). Sie werden durch einen ausgefüllten Kreis dargestellt.



Hybridparameter können sowohl Daten- als auch Kontrollparameter beschreiben, beispielsweise im Erfolgsfalle einen Wert, im Fehlerfalle einen Fehlerindikator. Diese Art von Parametern stellt aber eine schlechte Kopplungsart dar und sollte vermieden werden. Darauf werden wir später noch zu sprechen kommen.

Parameter mit einem Doppelpfeil stellen Returnwerte der Operation dar. Je nach Art des Ergebnisses wird der Kreis leer gelassen, ausgefüllt oder mit einem Punkt versehen. Eine Aufrufkante kann nur einen Returnparameter haben. Returnparameter sind immer Ausgabeparameter.

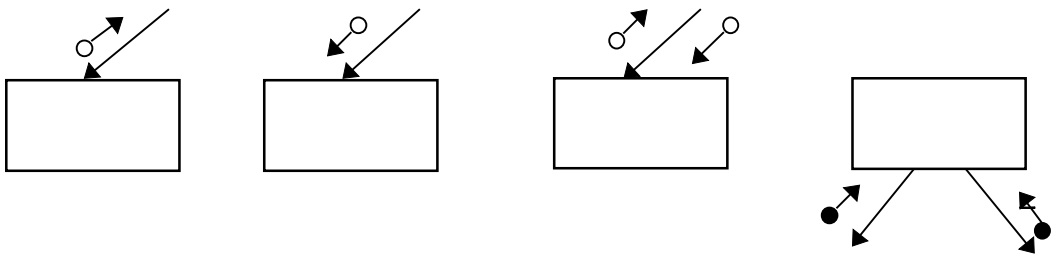
Die beiden letztgenannten Arten von Parameter, nämlich Hybridparameter und Returnparameter, werden wegen ihrer spezifischen Eigenheiten nicht in jeder Methode unterstützt. Sie kamen beispielsweise im ursprünglichen Structured Design nicht vor.

6.5.4 Beispiel eines Operationsdiagramms

ToDo

6.5.5 Arten von Operationen

Je nachdem wie eine Operation mit den anderen verknüpft ist, kann man verschiedene Klassen von Operationen unterscheiden:



Eingabe-Operation | **Ausgabe-Operation** | **Transformations-Operation** | **Koordinierungs-Operation**

6.5.6 Beschreibung von Operationen

Jede in einem Operationsdiagramm verwendete Operation muß auch beschrieben werden. Man unterscheidet drei Teile in einer solchen Beschreibung:

+ **Funktionsbeschreibung**
  d.h. nur bzgl. des Verhaltens an den Schnittstellen und nicht des inneren Aufbaus (black-box)

+ **Schnittstellenbeschreibung**
  während in den Diagrammen die aktuellen Parameter dargestellt sind, wird hier die Definition der formalen Parameter gegeben.

+ **Realisierungsbeschreibung**
  Beschreibung der internen Arbeitsweise eines Moduls, also die Algorithmenbeschreibung

Letzteres stellt den Übergang zur Implementierung her. Neben einer verbalen Form, die im wesentlichen die Dokumentation zum Source-Programm liefert, gibt es auch formalere Formen, etwa formal definierten Pseudocode oder Struktogramme, aus denen Werkzeuge auch direkt Code erzeugen können oder zumindest Codefragmente erzeugen können, die dann noch während der Implementierungsphase auszufüllen sind.

Bezüglich Pseudocode und Struktogramme (Nassi-Shneidermann-Diagramme) wird auf das Buch Goll/Grüner/Wiese: „C als erste Programmiersprache“ verwiesen.

6.5.7 Eigenschaften eines Operationsdiagramms

Operationen repräsentieren Funktionen oder Prozeduren einer Programmiersprache. Ähnlich wie in den Moduldiagrammen wird auch hier nur die statische Aufrufstruktur dargestellt und keine Rücksicht auf den dynamischen Programmablauf genommen.

Wichtigste Information ist neben der Aufrufbeziehung die Darstellung der Schnittstelle, d.h. die äußere (die Blackbox-) Sicht auf eine Operation. Eine Operation sollte in ihrem Umfang so bemessen sein, dass sie im Rahmen ihrer Realisierungsbeschreibung mit einem Struktogramm in übersichtlicher Weise dargestellt werden kann.

Das Diagramm läßt sich leicht automatisch aus dem Quelltext herleiten, jedoch sollte es selbstverständlich vor der Implementierung der Module erstellt werden! Der wichtigste Nutzen für die Implementierung besteht darin, dass aus der Schnittstellendefinition der Operation der Prototyp der Funktion abgeleitet werden kann.

6.6 Größeres Beispiel

ToDo

6.7 Entwurfsprinzipien

Die Prinzipien, nach denen geeignete Module gefunden werden können, bzw. die beim Erstellen der Hierarchien beachtet werden sollten, sind im wesentlichen:

- das Black-Box Prinzip,
- das Lokalitätsprinzip,
- das Geheimnisprinzip,
- das Kapselungsprinzip.

Diese sind eigentlich unabhängig von einer konkreten Methode und stellen grundlegende Designprinzipien dar.

6.7.1 Das Black-Box Prinzip

Beim Betrachten einer Black-Box findet man, daß die Eingaben und die Ausgaben bekannt sind, sowie die Funktionalität, das was getan wird. Man findet aber nicht eine Beschreibung, wie die Funktionalität realisiert ist.

Das Black-Box Prinzip in der Software-Entwicklung bedeutet:

- Es sollen Module/Operationen geschaffen werden, die jeweils eine wohldefinierte Funktion erfüllen.
- Die ausgeführte Funktion soll leicht verständlich sein.
- Die Black-Boxes dürfen keine Annahmen über die Implementierung anderer Black-Boxes machen.
- Die Anzahl der Schnittstellen zwischen den Black-Boxes soll minimal sein.
- Die Schnittstellen sollen einfach sein.

Die Vorteile, die sich daraus erzielen lassen sind, dass Black-Boxes

- einfach zu konstruieren,
- einfach zu verstehen,
- einfach zu testen,
- einfach zu korrigieren, zu modifizieren, kurz
- einfach zu warten

sind. Das Black-Box Prinzip ist unterstützt also in grundlegender Weise die für den Entwurf gesetzten Ziele.

6.7.2 Das Lokalitätsprinzip

Lokalität bedeutet, dass inhaltlich zusammengehörige Entwurfsentscheidungen auch im Modell eng benachbart sind. Entwurfsentscheidungen, die nichts miteinander zu tun haben, werden auch getrennt abgelegt.

Damit das Lokalitätsprinzip gewahrt bleibt, sind auch ein paar physikalische Regeln für das Modell zu beachten:

- Alle Dokumente sollen kleiner sein als eine DIN A4 Seite.

- Es sollen nur 5 +/- 2 Elemente auf einer Seite bzw. in einem Diagramm enthalten sein.

Module sollen zusammengehörende Funktionen zusammenfassen und nicht zusammengehörende trennen.

6.7.3 Das Geheimnisprinzip

Während das Lokalitätsprinzip sagt, dass jede wichtige Entwurfsentscheidung an genau einer Stelle durchgeführt werden soll, sagt nun das Gehimnisprinzip, dass diese Entscheidung auch von seiner Umgebung verborgen werden soll. Es muss nicht jeder über alles informiert sein. Man unterscheidet zwischen Information Hiding (logisch) und Implementation Hiding (physikalisch).

Alles, was ein Entwickler verbergen will, kann auch verborgen werden, beispielsweise:

- Wahl der Datenstrukturen (Stack als verkettete Liste, Feld oder File),
- Wahl eines Algorithmus (Gehaltszahlung),
- Interne Systemzustände (Task Control Block Bit 3 bedeutet: Task läuft),
- Entscheidungsstrukturen (Gehaltserhöhung),
- Daten (Drehwinkel eines Roboters),
- Eigenschaften von physikalischen Geräten (Blinken des Bildschirms).

Der Entwickler entscheidet, was verborgen wird und auf welche Weise. Für die Benutzung stellt er eine gut dokumentierte Schnittstelle zur Verfügung (Black-Box Prinzip).

Der Vorteil des Geheimnisprinzips ist, dass verborgene Informationen geändert werden können, ohne daß die Umgebung davon beeinflußt wird. Die Gründe für eine Veränderung von verborgenen Information sind vielfältig, beispielsweise Änderung der Technologie, Steigerung der Effizienz, Erweiterung/Verbesserung der Datenstrukturen.

6.7.4 Das Kapselungsprinzip

Kapselung (Encapsulation) bedeutet

- Zusammenfassen von ähnlichen Funktionen in eine logische Einheit, die wie ein einzelner Baustein benutzt werden kann.
- Kontrollierter Zugriff auf die logische Einheit durch genau definierte Schnittstellen.

Mit Hilfe der Kapselung wird also ein Objekt zur Black-Box. Ein außenstehender Beobachter sieht nur die äußerlich sichtbaren Eigenschaften und das äußerlich sichtbare Verhalten eines solchen Objekts. Er kann nicht sehen, wie die internen

Informationen strukturiert sind, wie interne Funktionen implementiert sind und wie sie sich verhalten.

Ohne Kapselung sind alle Daten global verfügbar und es kann auf jedes Statement im Code direkt zugegriffen werden. Mit Kapselung ist der Zugriff nur auf solche Objekte erlaubt, die explizit zugänglich gemacht wurden. Man erreicht damit eine höhere Flexibilität innerhalb einer gekapselten Einheit und erzielt eine strenge Zugriffskontrolle auf die gekapselte Einheit von außerhalb.

Kapselung ist die Kombination aus Information Hiding und Black-Boxes.

6.7.5 Anwendung: Transformation eines SA-Modells

Unter Beachtung dieser Designkonzepte können einige Transformationsregeln aufgestellt werden, mit deren Hilfe Modelle der SA in einen modularen Entwurf überführt werden können. Voraussetzung ist allerdings, dass ein entsprechendes Realisierungsmodell angestrebt wird, in diesem Falle das einer prozeduralen Programmiersprache.

- Das Kontextdiagramm wird in das Hauptprogramm transformiert.
- Jedes andere DFD wird zu einem Modul mit einer Exportfunktion.
- Verfeinerte Knoten (Sohnknoten) werden zu Importfunktionen im Modul zum Vaterdiagramm.
- Basisknoten (nicht weiter verfeinerte Knoten) werden zu internen Funktionen des Moduls, der ihrem Diagramm zugeordnet ist.
- Aus Speichern werden Moduln mit vier Exportfunktionen: Create, Read, Update und Delete.
- Die Definition eines Speichers wird zu einem Exportdatentyp während der Speicher selbst ein internes Datum des Moduls wird.
- Prozesse, die den Speicher benutzen, müssen diesen Modul importieren.
- Ein Datenfluss zwischen zwei Knoten wird zu einem Funktionsaufruf.

Diese Regeln hier erheben keinen Anspruch auf Vollständigkeit. Beispielsweise ist das ganze Gebiet RT und DD hier außen vor gelassen.

Es sollte auch klar sein, dass es immer irgendwelche Sonderfälle zu betrachten gibt, die eine andere Behandlung erfordern. Wäre dem nicht so, dann könnte man ja diese Transformationsregeln automatisieren und könnte direkt aus dem Modell Code generieren. Es gibt Werkzeuge, die eine Unterstützung anbieten, nicht als Automatismus, sondern als eine Art Assistent, der dem Designer beim Aufbau seiner Systemstruktur hilft.

Viele Werkzeuge bieten zumindest eine Verknüpfung zwischen dem SA Modell und den MD Komponenten an. Bei jeder MD Komponente (Modul, Operation, Daten) kann eine Zuordnung zu einem Element des SA Modells (Prozess, Speicher, Diagramm, Terminator, ...) erfolgen, was bedeuten soll, diese MD Komponente ist verantwortlich für die Einhaltung der im Modell für das betreffende Element festgelegten Anforderungen. Damit können Werkzeuge zumindest prüfen, ob die MD Struktur das SA Modell vollständig umsetzt.

6.7.6 Zusammenfassung

Die Beachtung der allgemeinen Designkonzepte ist die Grundlage um die Designziele zu erreichen. Dann werden automatisch Systeme erstellt, die wartbar, flexibel und anpassbar sind. Die Komponenten eines solchen Systems sind in einem hohen Maße wiederverwendbar.

Die daraus resultierenden Eigenschaften der Module sind im Wesentlichen:

- Er kann ohne Kenntnis seines inneren Aufbaus in eine Umgebung eingebettet werden.

- Er kann ohne Kenntnis der Umgebung, in die er eingebettet werden soll, entworfen und implementiert werden.

- Er soll nur logisch zusammenhängende Aufgaben beinhalten.

- Seine Schnittstellen sollen möglichst einfach und schmal sein, insbesondere aber eindeutig sein.

- Er soll möglichst getrennt übersetzbar sein, damit der Aufbau von Modul-Libraries gefördert wird.

Existiert bereits ein (Bibliotheks-) Modul, welches die gewünschte Funktion erfüllt, bzw. nach geringer Modifikation erfüllen könnte, dann sollte eine Schnittstelle so definiert werden, dass das bestehende Modul übernommen werden kann.

Falls notwendig sind Entwurfsentscheidungen zu treffen und hinreichend in den entsprechenden Modulspezifikationen zu begründen. Entwurfsentscheidungen können z.B. Betriebssystem, Implementierungssprache, Datentypen für Schnitt-stellen,... betreffen.

6.8 Evaluation eines Entwurfs

Der Entwurf legt die Grundlage für

- **hochgradig paralleles Vorgehen** innerhalb der Implementierungsphase
  Dadurch werden kürzere Fertigungsdauern realisiert.

- **unabhängige Realisierung** von Teilaufgaben
  Durch klar definierte Schnittstellen und Modulspezifikationen ist keine Abstimmungen mehr erforderlich.

- **Wiederverwendung der Software**
  Durch den modularen Aufbau ist Austauchbarkeit von Modulen gegeben.

Die Evaluation eines Entwurfs ist besonders wichtig, weil Fehler innerhalb dieser Phase erst nach der Implementierung festgestellt werden, nämlich zum Zeitpunkt der Integration. Die Güte eines Entwurfs zeigt sich manchmal sogar noch später erst, nämlich bei der Wartung oder wenn die Moduln wiederverwendet werden sollen.

6.8.1 Entwurfsvalidierung

Grundsätzlich muß daher eine Überprüfung des Entwurfs erfolgen bezüglich:

+ **Vollständigkeit**

+ **Widerspruchsfreiheit** (Konsistenz).

Konkret heißt das, daß folgende Fragen geprüft werden müssen:

- Enthält Hierarchiediagramm bzw. Structure-Chart alle Module/Operationen, welche zur Lösung der Aufgabe benötigt werden?

- Wurden alle im Hierarchiediagramm bzw. den Structure Charts auftretende Module/Operationen in einer Modulbeschreibung explizit definiert?
  - Spezifikation der Aufgabe
  - Spezifikation der Schnittstelle

- Existiert für jede Operation mindestens eine Aufrufstelle in einer anderen Operation oder in sich selbst (Rekursion)?

- Werden die Schnittstellen im definierenden und angewandten Auftreten einer Operation konsistent verwendet?
  - gleiche Anzahl von Parametern
  - gleiche Reihenfolge der Parameter
  - gleicher Typ (auch ob in, out oder in/out)

- Wurden nicht vermeidbare globale Daten hinreichend beschrieben und nur den Modulen bekannt gemacht, welche sie verwenden?

• Stehen alle benötigten Bibliotheksfunktionen tatsächlich auf Abruf bereit?
- Wurden sie in früheren Projekten korrekt implementiert?
- Verursachen / basieren sie auf speziellen Seiteneffekten?

Entwurfsvalidierung kann manuell oder in Grenzen auch maschinell (bei vorhandenem CASE-Tool) erfolgen.

. Merke:

> Ein CASE-Tool kann lediglich formale Aspekte der Spezifikation prüfen, die Semantik bleibt ihm verborgen!

6.8.2 Bewertung eines Entwurfs

Um die Komplexität von Schnittstellen und die funktionale Vollständigkeit beurteilen zu können gibt es zwei Kriterien: die Kopplung (Coupling) und den Zusammenhalt (Cohesion).

- Kopplung ist ein Maß für die Unabhängigkeit der Moduln und Operationen untereinander. Der Grad der Unabhängigkeit kann durch Untersuchen der Schnittstellen festgestellt werden. Es ist ein Entwurfsziel Moduln zu erstellen, die nur schwach gekoppelt sind (loose coupling).

- Zusammenhalt ist ein Indikator für die Komplexität eines Moduls und damit für die Verständlichkeit. Zusammenhalt ist ein Maß für die Stärke des funktionalen Zusammenhangs der Elemente eines Moduls. Es ist ein Entwurfsziel Moduln zu erstellen, die einen hohen Zusammenhalt aufweisen (high cohesion).

Diese beiden Kriterien werden nun im Folgenden vorgestellt. Danach werden die Methoden skizziert, die einem helfen diese Ziele zu erreichen: Faktorisieren von Moduln, Überlegungen zum Fan-In/Fan-Out eines Moduls und Begrenzung des Scope-Of-Effects.

6.8.3 Kopplung von Modulen

Kopplung (Coupling) ist ein Maß für die Unabhängigkeit der Moduln und Operationen untereinander. Der Grad der Unabhängigkeit kann durch Untersuchen der Schnittstellen festgestellt werden.

Es ist ein Entwurfsziel Moduln zu erstellen, die nur schwach gekoppelt sind (loose coupling). Die Minimierung der Kopplung ist notwendig, da bei hoher Kopplung die folgenden Probleme auftreten:

- Es ist unmöglich, eine Funktion zu verstehen, ohne dass noch andere Funktionen mitbetrachtet werden müssen.
- Von Änderungen/Erweiterungen sind immer viele Funktionen betroffen.

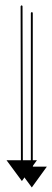
- Es gibt eine hohe Wahrscheinlichkeit, dass durch eine Änderung eine Funktion mit hoher Kopplung ebenfalls betroffen ist.

Je weniger Kopplung zwischen zwei Funktionen herrscht, umso unabhängiger sind diese Funktionen. Je unabhängiger die Funktionen sind, umso besser ist der Entwurf. Eine völlige Vermeidung von Kopplung ist nicht möglich.

Durch Minimierung der Schnittstellenbandbreite wird ein Design verbessert. Ziel dabei ist die kleinste notwendige Menge an Daten zwischen Funktionen auszutauschen. Die Aufteilung in mehrere Moduln/Funktionen, um den Umfang der Schnittstelle zu minimieren, verbessert (verringert) gleichzeitig die Kopplung. Kann die Schnittstelle nicht weiter verbessert werden, hilft noch die Betrachtung der Kopplungsart.

Man unterscheidet folgende Kopplungstypen:

- Daten
- Datenstruktur
- Signal
- Global
- Inhalt

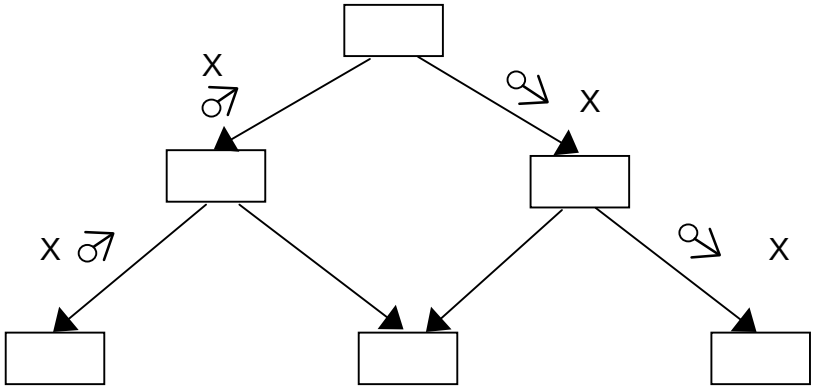


gut

schlecht

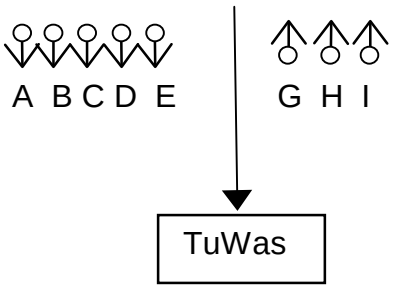
Zwei Funktionen sind mittels **Datenkopplung** verbunden, wenn die eine Funktion die andere aufruft und die Kommunikation zwischen ihnen durch Datenparameter erfolgt. Datenkopplung stellt damit die notwendige Kommunikation zwischen Funktionen dar und ist daher unvermeidbar. Trotzdem sollte sie auf ein Minimum reduziert werden.

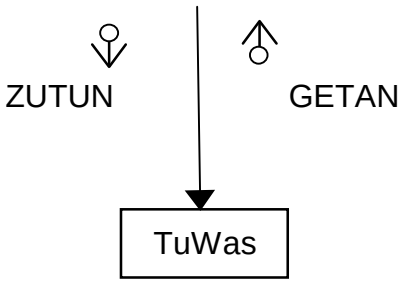
Die Verwendung von Datenkopplung ist keine Garantie für einen guten Entwurf. Beispielsweise sollte die Datenwanderung vermieden werden, bei der Daten ohne Benutzung durch Funktionen hindurchwandern.



Zwei Funktionen sind durch **Datenstrukturkopplung** verbunden, wenn sie mit mindestens einem Element aus einer Datenstruktur (Feld, Record) miteinander kummunizieren. Bei hoher Datenkopplung ist eine sinnvolle Zusammenfassung zu einer Datenstruktur eine gute Anwendung von Datenstrukturkopplung.

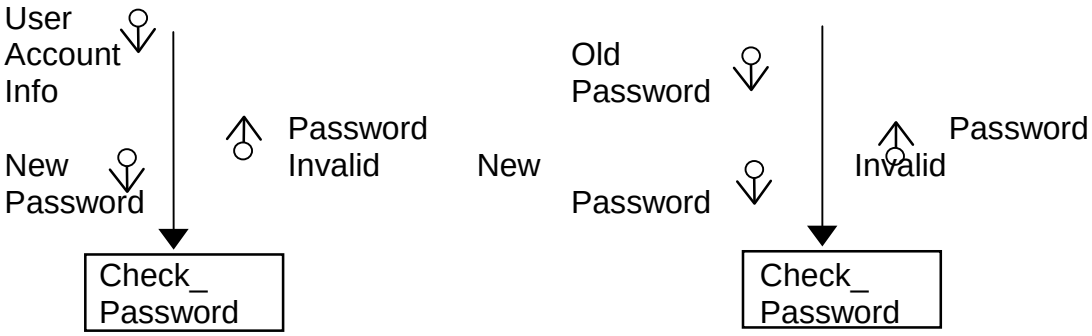
Man sollte allerdings nicht einfach nur Daten, die nichts miteinander zu tun haben, in einer Datenstruktur zusammenfassen, um damit die Schnittstellenbandbreite zu verringern.





ZUTUN = A+B+C+D+E
GETAN = G+H+I

Datenstrukturkopplung sollte vermieden werden, wenn es möglich ist mit einer überschaubaren Menge von Datenkopplungen auszukommen. Dazu werden folgende Alternativen betrachtet:



Bei der Datenstrukturkopplung können folgende Probleme auftreten:

1. Eine Änderung in UserAccountInfo kann die Funktion Check_Password beeinflussen, selbst wenn die Änderung nichts mit Passwörtern zu tun hat.
2. Die Funktion Check_Password erhält Daten, die sie nicht braucht (Information Hiding verletzt).
3. Ein Fehler in der Funktion Check_Password könnte Daten in UserAccountInfo modifizieren, was aber dann in einer ganz anderen Funktion zu Tage tritt.

Man sieht also an diesem Beispiel, dass Datenkopplung zwar zu einer größeren Schnittstelle führt, aber andere Vorteile gegenüber der Datenstrukturkopplung bietet.

Zwei Funktionen kommunizieren mittels **Signalkopplung**, wenn mindestens ein Steuersignal ausgetauscht wird. Durch Signalkopplung wird das Verhalten des Empfängers beeinflußt. Dies ist grundsätzlich unerwünscht, da eine der beiden Funktionen dann keine Black-Box mehr ist. Je mehr Entscheidungsfreiheit die Empfängerfunktion hat, umso mehr ist das Black-Box Prinzip gewahrt.

Signalkopplung tritt in zwei Formen auf:

1. Das Signal trägt Information, wie etwa den Status, Plausibilität, EOF etc. Damit ist die Entscheidungsfreiheit des Empfängermoduls gewahrt.
2. Das Signal enthält die Information, wie der Empfängermodul weitermachen soll, etwa in Form von Befehlsadressen, Marken, Funktionen, die aufgerufen werden sollen etc. Dies tritt bespielsweise auf bei sogenannten Callback-Mechanismen.

Bei hybrider Kopplung wird Signal- und Datenkopplung gemischt. So können Schlüsselfelder einer Tabelle auf der einen Seite Daten identifizieren (Datenkopplung) auf der anderen Seite auch Information tragen, die das Verhalten eines Moduls beeinflußt, wie im folgenden Beispiel:

Kundenschlüssel	0000-8899:	Standard Kundenkonto
	8900-8999:	Internes Einkaufskonto
	9000-9999:	Mitarbeiter Einkaufskonto

Hybride Kopplung sollte vermieden werden, indem man den Daten- und den Signalanteil trennt. Dadurch wird zwar wieder die Schnittstellenbreite erhöht, was eigentlich schlecht ist, dafür aber gewinnt die Klarheit der Schnittstelle enorm.

Funktionen sind **global gekoppelt**, wenn sie Daten gemeinsam benutzen, die nicht über einen normalen Funktionsaufruf ausgetauscht werden. Damit wird effektiv die eigentliche Funktionskommunikation verborgen, die die Grundlage für das Prüfen und Verbessern eines Entwurfs ist. Die Probleme der Globalen Kopplung sind:

- Funktionen, die globale Daten benutzen, greifen darauf über einen expliziten Namen zu. Eine Wiederverwendung wird damit schwierig.
- Unterschiedliche, nicht zusammenhängende Daten, die in einem gemeinsamen (globalen) Speicherbereich untergebracht sind, sind schwer zu warten.
- Die Verwendung von globalen Datene erschwert die Lesbarkeit eines Programms. Es ist kaum möglich die Kopplungsart zu bestimmen, wenn die Kommunikation im wesentlichen über globale Daten erfolgt.
- Außerdem ist es schwer zu sagen, welche Funktion welche Daten benützt. Bei Änderung an einer Stelle, müssen alle Funktionen geprüft werden, um sicherzustellen, dass keine Seiteneffekte aufgetreten sind.
- Fehler in einer Funktion treten in der Regel an ganz anderen Stellen im System zu Tage und erschweren damit ebenfalls die Wartung.

Wenn Funktionen beispielsweise aus Effizienzgründen nicht über Parameter kommunizieren sollen, sondern global gekoppelt sein sollen, dann sollte man wenigstens alle Funktionen, die den globalen Datenbereich nutzen, in einem Modul zusammenfassen und den globalen Datenbereich in diesem Modul verbergen, sodass er außerhalb nicht mehr zugreifbar ist. Dadurch werden einige der vorherigen Argumente entkräftet, das Lokalitätsprinzip und das Kapselungsprinzip bleiben gewahrt.

Inhalts-Kopplung ist eine Kopplungsart, die unbedingt vermieden werden muss. Man versteht darunter, dass sich eine Funktion auf die interne Struktur einer anderen Funktion bezieht, beispielsweise dass eine Funktion eine Anweisung einer anderen

Funktion ändert, sich auf interne Daten einer anderen Funktion bezieht, oder mitten in eine andere Funktion verzweigt.

Man nennt diese Art der Kopplung auch pathologisch. Die meisten heutigen Programmiersprachen lassen diese Art der Kopplung auch gar nicht mehr zu. In Assembler sind solche Kopplungsarten allerdings durchaus möglich, ebenso bei Programmiersprachen, die sich selbst modifizierende Programme erlauben.

Zusammenfassend läßt sich sagen:

- Je loser die Kopplung zweier Funktionen, desto unabhängiger sind sie.
- Im Zweifel wählt man die schlechtere Kopplungsart (um sie dann zu verbessern).

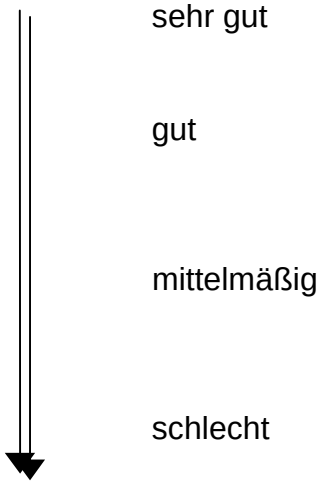
Man betrachtet seine Funktionen wie Bibliotheksroutinen und fragt: Wie erhalte ich den größten Nutzen? Wie verbessere ich die Wartbarkeit?

6.8.4 Zusammenhalt eines Moduls

Der Zusammenhalt (Kohäsion, Cohesion) ist ein wichtiger Indikator für die Komplexität eines Moduls und damit für die Verständlichkeit. Kohäsion ist ein Maß für die Stärke des funktionalen Zusammenhangs der Elemente innerhalb eines Moduls. Module mit hoher Kohäsion sind leicht verständlich und damit besser wartbar.

Die Elemente einer Funktion sind Anweisungen, Gruppen von Anweisungen oder Aufrufe an andere Funktionen. Die Elemente eines Moduls im Modular Design sind Funktionen und zentral definierte Daten (Konstanten, Datentypen).

Für die Bewertung des Zusammenhalts gibt es die folgenden Kriterien:

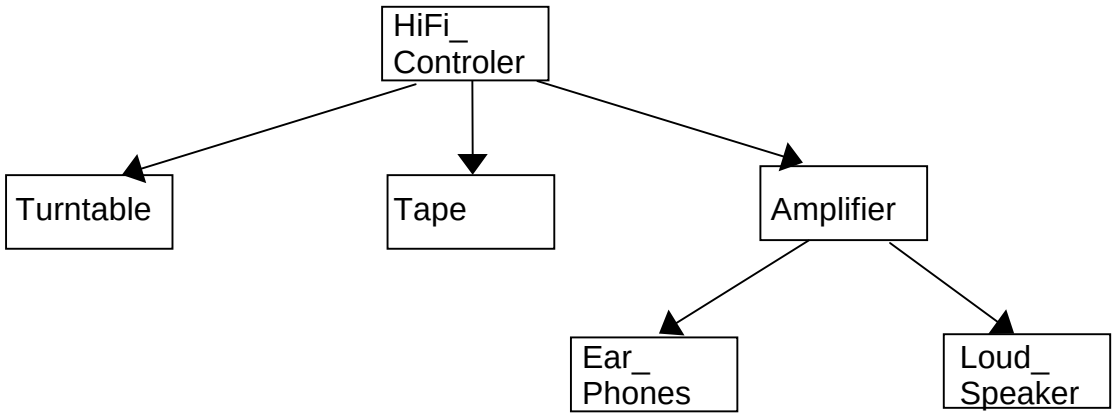


Ein Modul ist **funktional kohäsiv**, wenn alle Elemente der Erfüllung einer einzigen Aufgabe dienen. Daher lautet eine häufige Forderung, dass man die Aufgabe eines Moduls in einem Satz zusammenfassen können müsse.

Die folgenden Elemente des Moduls "Wandle Grad in Bogenmaß um" sind funktional kohäsiv:

"Wandle ASCII-Eingabe in internes Format um"
"Dividiere die Eingabe durch 360 und multipliziere Ergebnis mit Pi"
"Wandle Ergebnis in einen ASCII Zeichenstring um"

Die Beurteilung eines Moduls hängt vom Standpunkt des Moduls ab: Moduln die von einem Modul gerufen werden, sollten funktional kohäsiv sein. Moduln, die andere Moduln benutzen, erscheinen oft weniger als funktional kohäsiv. Betrachten wir dazu das folgenden Beispiel:



Aus der Sicht des HiFi-Controllers sollte das Modul Amplifier funktional kohäsiv sein. Aus der Sicht des Moduls Amplifier bedient das Modul HiFi-Controller noch andere Moduln. Es erscheint damit weniger als funktional kohäsiv.

Ein Modul ist **sequentiell kohäsiv**, wenn die Reihenfolge der Ausführungen entscheidend ist. Es findet eine aufeinanderfolgende Verarbeitung von Daten statt. Die einzelnen Elemente des Moduls müssen nacheinander aufgerufen werden.

Die folgenden Elemente des Moduls "Lackiere ein Auto neu" sind sequentiell kohäsiv:

Auto waschen
Fenster und Chromteile abkleben
Löcher ausfüllen
Auto sandstrahlen
Rostgrund auftragen
trocknen
Lack auftragen
trocknen
polieren

Ein Modul ist **kommunikativ kohäsiv**, wenn für die Durchführung verschiedener Aufgaben die gleichen Ein- oder Ausgabedaten benutzt werden.

Die folgenden Elemente des Moduls "Buchbearbeitung" sind kommunikativ kohäsiv:

- suche Buchtitel
- suche Buchpreis
- lese Buch
- schreibe Buchzusammenfassung
- bestimme Wörter pro Seite
- suche ISBN-Nummer

Kommunikative Kohäsion ist akzeptabel, da die Funktionen eines Moduls wegen ihrer Datenstruktur zusammengefaßt sind. Ein Modul mit kommunikativer Kohäsion kann häufig aufgeteilt werden in einzelne Moduln, wodurch die Kopplung vereinfacht werden kann und der Zusammenhalt verbessert werden kann.

Ein Modul ist **prozedural kohäsiv**, wenn die Elemente in einer bestimmten Reihenfolge ausgeführt werden müssen, aber im Gegensatz zur sequentiellen Kohäsion keine gemeinsamen Daten zu bearbeiten sind. Man erhält solche Moduln, indem man das Ablaufdiagramm eines Problems zerschneidet.

Die folgenden Elemente des Moduls "Bereite Weihnachtsessen vor" sind prozedural kohäsiv:

- Küche aufräumen
- Truthahn bratfertig zubereiten
- Truthahn in den Backofen schieben
- Duschen gehen
- Gemüse putzen
- Tisch decken
- Familie zu Tisch bitten
- Truthahn servieren

Bei einer **zeitlichen Kohäsion** müssen die einzelnen Elemente nicht mehr in einer bestimmten Reihenfolge durchlaufen werden, sondern sie müssen nur noch zur "gleichen Zeit", also in einem zeitlichen Zusammenhang, aber in beliebiger Reihenfolge ausgeführt werden.

Die folgenden Elemente des Moduls "Morgenzeremonie" sind zeitlich kohäsiv:

- ins Badezimmer gehen
- Kaffeemaschine anstellen
- Zeitung und Milch ins Haus holen
- Tisch decken

Typische Beispiele in der Datenverarbeitung sind Initialisierungsmoduln.

Ein Modul hat **logische Kohäsion**, wenn die Elemente ähnliche Aufgaben, aber mit völlig verschiedenen Ein- oder Ausgabedaten durchführen.

So sind etwa die folgenden Elemente des Moduls "Reisen" logisch kohäsiv:

reise mit dem Auto
reise mit dem Zug
reise mit dem Flugzeug
reise mit dem Schiff

Ein solches Modul ist meist dadurch gekennzeichnet, dass viele Flags und Schalter gesetzt werden, um die unterschiedlichen Daten zu bearbeiten. Dadurch kann ein solches Modul so unübersichtlich werden, dass niemand mehr weiß, was tatsächlich passiert. Das ist der Grund, warum diese Kohäsion so tief eingestuft wird, trotz des Namens, der eigentlich auf eine gute Eigenschaft hindeutet. Daher sollte es besser unlogische Kohäsion heißen.

Module sind **zufällig kohäsiv**, wenn keine andere Kategorie zutrifft. Die Vermutung ist, dass die Elemente zufällig zu einem Modul zusammengefasst wurden. Module mit zufälliger Kohäsion entstehen manchmal, wenn gemeinsame Stücke von Code entdeckt werden.

So sind etwa die folgenden Elemente des Moduls "Verschiedenes" zufällig kohäsiv:

Haus streichen
Kaffee trinken
Völkerball spielen
Auto waschen

Zur Bestimmung der Art der Kohäsion kann man folgende Heuristiken anwenden:

- Man versucht einen einzigen Satz zu schreiben, der die Modulfunktion vollständig und genau beschreibt.
- Falls dieser Satz ein Satzgefüge ist mit einem Komma oder mehr als einem Verb, dann ist der Modul wahrscheinlich weniger als funktional.
- Falls dieser Satz ein Verb enthält mit einem oder mehreren "und", dann ist der Modul wahrscheinlich sequentiell, kommunikativ oder prozedural.
- Falls dieser Satz Zeitwörter enthält wie "zuerst", "als nächstes", "danach" etc., so ist der Modul wahrscheinlich prozedural.
- Falls der Satz mehr als ein Verb hat mit Verwendung von "oder", so ist der Modul wahrscheinlich logisch.
- Falls der Satz komplex ist und mehrere "und" oder/und "oder" enthält, so ist der Modul wahrscheinlich zufällig.

Man kann nicht unbedingt erwarten, dass man die Kategorie exakt bestimmen kann. Im Zweifelsfall ordnet man den Modul einer schlechteren Kategorie zu.

Je besser die Kohäsion ist, desto geringer ist das Risiko, dass eine Spezifikationsänderung eine große Anzahl von anderen Moduln betrifft, desto lokaler sind die Änderungen begrenzt. Insgesamt wird das System einfacher wartbar und damit kostengünstiger.

Interessanterweise stehen Kopplung und Kohäsion miteinander im Zusammenhang: Bessere Kohäsion führt automatisch dazu, dass die Kopplung abnimmt.

6.8.5 Faktorisieren von Moduln

Das Herauslösen von Teilen eines Moduls in ein eigenes Modul bezeichnet man auch als Faktorisierung. Dies kann eingesetzt werden, um

- die Modulgröße zu reduzieren,
- Redundanz zu vermeiden,
- Ausführung und Steuerung zu trennen.

Die Faktorisierung sollte nicht mehr weitergeführt werden, wenn die Schnittstelle eines Moduls komplexer wird als die des alten Moduls. oder wenn man keine weiteren Untermoduln mehr findet, bzw. wenn die Moduln einfach genug sind, um sie leicht zu verstehen. Die physikalischen Merkmale, die man hier zu Rate ziehen kann, wurden bereits genannt: kein Diagramm sollte größer als eine DIN A4 Seite sein und kein Diagramm sollte mehr als 7 +/- 2 Elemente enthalten.

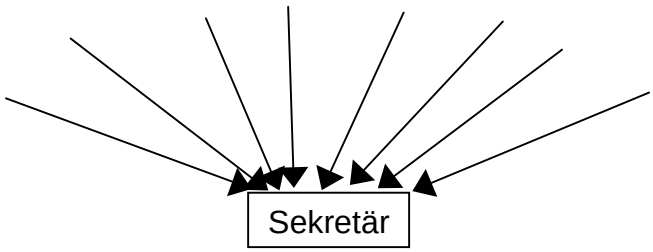
Die Faktorisierung sollte aber betrieben werden, weil

- das System danach leichter zu verstehen ist, und
- Veränderungen im System einfacher durchzuführen sind (lokal begrenzt).

Herausfaktorisierte Moduln stellen häufig allgemein verwendbare Funktionen dar, die mehrfach in verschiedenen Systemteilen benutzt werden können. Damit wird also die Wartung von Duplikaten reduziert und es entstehen gute Kandidaten für die Wiederverwendung in anderen Projekten.

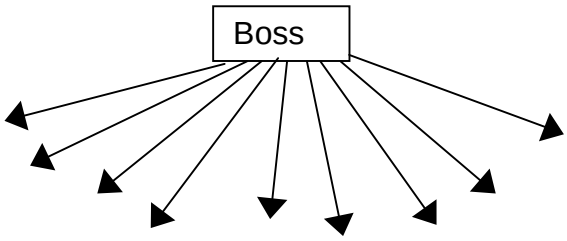
6.8.6 Fan-In/Fan-Out eines Moduls

Der Fan-In eines Moduls ist die Anzahl der übergeordneten Moduln, die es benutzen.



In diesem Fall ist der Fan-In des Moduls Sekretär 8. Im Design wird versucht den Fan-In zu maximieren. Ein hoher Fan-In bedeutet nämlich, dass es sich um einen häufig verwendeten, also wahrscheinlich auch allgemein verwendbaren, Modul handelt.

Der Fan-Out eines Moduls ist die Anzahl der direkt von ihm aufgerufenen Moduln.

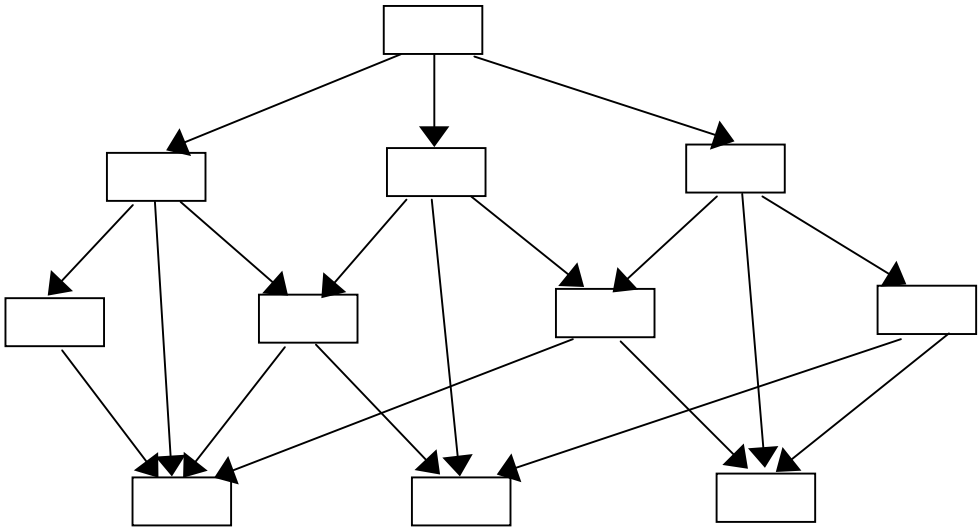


In diesem Fall ist der Fan-Out des Moduls Boss 9. Im Design wird versucht den Fan-Out eines Moduls zu minimieren. Er sollte auf alle Fälle auf 7 +/- 2 beschränkt sein. Ein hoher Fan-Out bedeutet, dass es sich um einen Modul handelt, der eine komplexe Logik beinhalten muss, um die vielen Module aufzurufen. Mit großer Wahrscheinlichkeit ist damit die Verständlichkeit und Wartbarkeit gering. Außerdem ist es unwahrscheinlich, dass der Modul noch anderweitig wiederverwendet werden kann, da er zu viele Abhängigkeiten hat. Ein geringes Fan-Out kann durch Faktorisieren erreicht werden.

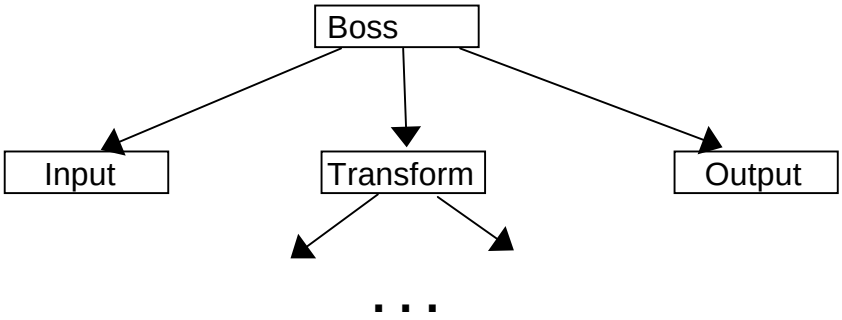
Ziel dieser Überlegungen ist ein **balanciertes System**. Ein System ist balanciert, wenn

- ein hoher Fan-Out auf den oberen Ebenen und
- ein hoher Fan-In auf den unterene Ebenen herrscht,

wie es das folgende Schema zeigt. Durch diese Bedingungen wird gewährleistet, dass auf den oberen Ebenen Daten von hoher Abstraktion verarbeitet werden.



Balancierte Systeme sind sehr stabil gegenüber Änderungen der Spezifikation oder externer Geräte. Im Gegensatz dazu stehen input- bzw. output-gesteuerte Systeme, bei denen die oberen Hierarchieebenen direkt mit „physikalischen“ Ein- oder Ausgabedaten arbeiten, wie es das folgende Bild zeigt.



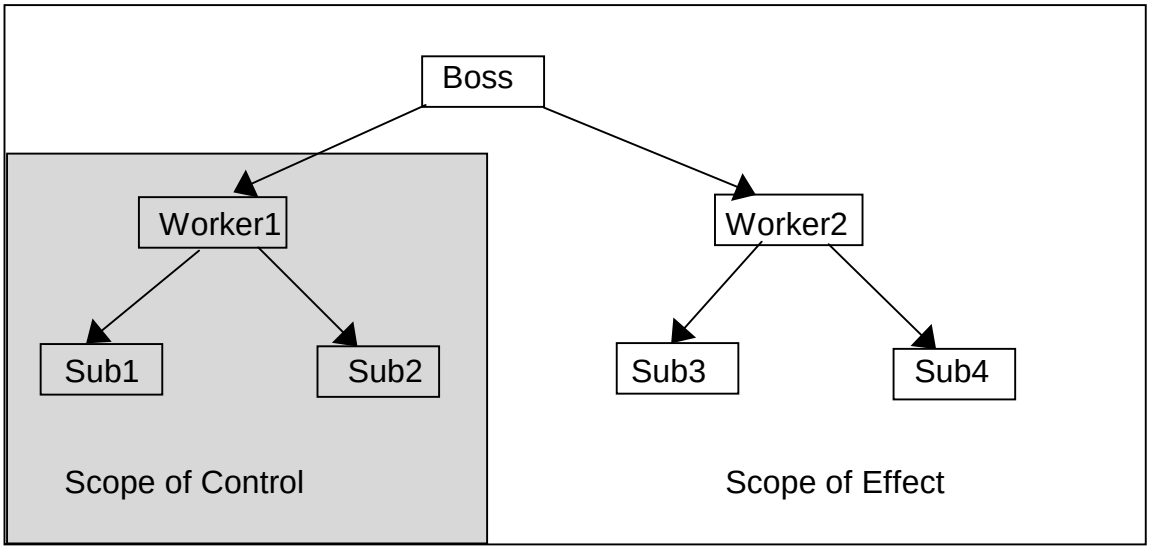
Input- oder output-gesteuerte Systeme sind sehr anfällig gegenüber Spezifikationsänderungen und können in der Wartungsphase sehr hohe Kosten verursachen. Sie sollten daher gemieden werden.

6.8.7 Scope-Of-Effect eines Moduls

Eine andere Überlegung, die man zur Bewertung eines Moduls anstellen kann, betrifft das Scoping. Man unterscheidet zwei Scopes:

- Der **Scope-of-Control** eines Moduls ist das Modul selber und alle seine untergeordneten Moduln.
- Im **Scope-of-Effect** eines Moduls liegen alle Moduln, die von einer Änderung einer Entwurfsentscheidung im Modul betroffen sind.

Die Modulhierarchie sollte so aufgebaut sein, dass jede Entscheidung eines Moduls nur sich selbst und seine untergeordneten Moduln betrifft (Lokalitätsprinzip und Geheimnisprinzip verbunden mit dem Scoping). Wenn dem so ist, dann liegt der Scope-of-Effect eines Moduls in seinem Scope-of-Control! Genau das ist das Ziel im Design, das an folgendem Bild schematisch dargestellt wird:



Der Scope-of-Effect eines Moduls sollte also eine Untermenge seines Scope-of-Control sein.

6.8.8 Zusammenfassung

Die beiden wichtigsten Kriterien für die Bewertung eines Entwurfs sind die Kopplung und der Zusammenhalt. Sie können auf alle Designelemente, also Operationen, Moduln und Packages angewandt werden.

Diese Ziele werden unterstützt durch den Einsatz von Faktorisierung, Begrenzung des Scope-of-Effects auf den Scope-of-Control, Vermeidung von zu großem Fan-Out und Verbesserung des Fan-In.

Voraussetzung (selbst bei kleinen SW-Projekten) ist stets:

Mindestens eine weitere Person sollte den Entwurf wirklich verstehen und ihn für korrekt halten!

7 Implementierung

7.1 Vom Entwurf zum Programm

Aus der Entwurfsphase liegt die Dekomposition der Problemlösung in Form einer Anzahl von Modulspezifikationen vor.

Sie beinhalten jeweils als wichtigste Elemente:

- Ein/Ausgabe-Schnittstelle
- Funktion / Aufgabe des Moduls
- Testdaten

Jedes Modul wird dabei durch einen Modulkopf (Modulheader) beschrieben und eingeleitet, etwa der folgenden Art:

```
Filename / Source: get_title.c

Modulname: get_title(...)                                   Version: 0.1

Autor: Frank Hack, hack@rhds01.fht-esslingen.de              Datum: 23.3.94

Modifiziert von: Alfred Mueller, Tel.: 3774                  Datum: 29.5.94
Modifiziert von: Fred Stiller, stiller@stone.mit.edu         Datum: 4.6.94
•••
Zustand: in Bearbeitung / Alpha-Test / Beta-Test / freigegeben / ...

Aufruf: get_title( server, kennung, destination)

Kurzbeschreibung:
liefert bei Angabe des Servers, der Kennung eines Eintrags und eines Zeigers
auf ein ausreichend großes Speicherstück den gewünschten Datensatz

Seiten-Effekte / globale Auswirkungen:  keine

Parameterbeschreibung:
server              (i)       Name des Rechners als Pointer auf String
                              z.B. als IP-Adr.: "134.108.3.32"
                              oder als Name: "rhds01"
•••
name                i/o/io    Bedeutung
```

Praktische Hinweise:

+ Ein entsprechender Header ist für jedes Modul im Rahmen des Entwurfs (spätestens als erster Schritt der Implementierung) zu erstellen.

+ Handelt es sich um ein globales Modul (z.B. Include/Makro), so sind unter dem Punkt *Seiteneffekte* alle Files anzugeben, welche das Makro verwenden!

+ Im Rahmen der Implementierung erfolgt die Erstellung eines Funktions-Prototypen, damit für das Konfigurationsmanagement bereits ein Modul (wenn auch noch unausgefüllt) zur Verfügung steht. Eventuell kann ein Dummy-Ergebnis zurückgeliefert werden, solange die eigentliche Version noch nicht existiert.

7.2 Implementierungsrichtlinien

Ziel ist eine möglichst "normierte" Programmierung zur

- schnellen Wiedererkennung gleichartiger Strukturen / Darstellungen
- Unterstützung bei der späteren Analyse und Einarbeitung in das Programm
- Unterstützung der späteren Programmvalidierung (Walkthrough) und Test

Damit wird gleichzeitig die Wiederverwendbarkeit von Modulen unterstützt. Der Nachteil besteht in einer (z.T. erhebliche) Einarbeitungszeit für neue Mitarbeiter.

Implementierungsrichtlinien können umfassen:

+ Textuelle Gestaltung des Source-Codes

- maximal 80 Zeichen je Zeile (stets darstellbar auf Terminal, Drucker)
- Blöcke werden durch Leerzeilen getrennt
- innerhalb von Blöcken gibt es keine Leerzeilen
- geschachtelte Blöcke werden je um 2 Spalten eingerückt
- Form der Klammerungen mit begin/end, do/od, {/}
- möglichst Einsatz von **sprachsensitiven** Editoren
  - automatisches Einfügen eines generischen Modulkopfes
  - automatisches Einrücken bei Blöcken
  - automatische Plazierung von

end	nach Eingabe von	**begin**
until		**repeat**
}		**{**

  - automatische Einfügung von Kommentar bei
    Variablendeklaration
    Klassendeklaration
    Funktionsdeklaration

+ Reihenfolge der Bestandteile eines Modulkopfes

- Name des Quelltext-Files
- Modulname
- Autor des Moduls
- Modifikationshistorie
- aktueller Zustand des Moduls
- Aufrufsyntax
- Funktionsbeschreibung des Moduls
  - kurz und prägnant, falls ausführliche Modulspezifikation an anderer Stelle
  - vollständige Spezifikation, falls es die einzige Stelle ist, an der die Aufgabe/Funktion des Moduls beschrieben wird
- Seiteneffekte bzw. globale Auswirkungen des Moduls
- genaue Parameterbeschreibung

+ Reihenfolge der Bestandteile eines Moduls

- Modulkopf
- eingeschobene / importierte Teile (#include, extern ...)
- selbst definierte Makros (z.B. #define)
- vereinbarte Konstanten
- vereinbarte Datentypen
- vereinbarte Datenstrukturen (Variablen)
- vereinbarte Unterprogramme und Funktionen

+ Namenskonvention für Bezeichner

- Namenslänge
  - möglichst selbsterklärend (oft länger)
  - falls Beschränkung, dann hier angeben (Π Compiler, Linker,...)
- Groß- / Kleinschreibung von Bezeichnern
  - Datentypen
  - Variablen
  - Prozeduren / Funktionen
  - Klassen
  - Objekte
- Komposition von Bezeichnern
  - Zusammensetzung von Teilnamen
  - Trennzeichen (z.B. ".", "_", "$",... fest vorschreiben)

Beispiel: Was bedeuten im folgenden Programm die Namen, was berechnet das Programm?

```
program CF (input, output);
var c,f : real;
begin
  read(c);
  f := c * 9 / 5 + 32;
  write(f);
end.
```

[Som 87]

+ Programmkommentar

- Dokumentation des laufenden Programmtextes (**Programmdokumentation**)
  - Form und Inhalt
  - Plazierung
- bei komplexen arithmetischen / logischen Operationen
- bei Aufrufen die Bedeutung der Parameter (in/out)
- bei Datendeklarationen, Objektinstanziierung
- bei Anweisungsblöcken oder mehreren Zeilen mit gemeinsamer Funktion

Hinweise für die Kommentierung:

- evtl. verschiedene Ebenen von Kommentaren zur leichteren Selektion
- eine Kommentierung von einzelnen Zeilen ist wenig sinnvoll
- eine Regel der Art mind. 50 % Kommentar ist wenig sinnvoll, weil

  bei 20% und mehr Kommentar keine Übersicht und keine Aussagekraft
  zu viel Kommentar zerstört die Programmstruktur
  größere Menge Kommentar wird bei Änderungen kaum aktuell gehalten

. Für die Dokumentation des Source-Codes gilt allgemein:
Gute Programmierer zeichnen sich durch einfache und klare Programme aus, **nicht** durch schwerverständliche Tricks und Kniffe.

. Die Dokumentation des Quell-Codes erfolgt
gleichzeitig mit der Codierung und durch den Programmierer selbst
d.h. nicht später und auch nicht von anderen Projektmitgliedern!

+ Programmierempfehlungen

- Einschränkung der verfügbaren Sprachelemente (∏ Portierbarkeit)
- keine Nutzung impliziter Definitionen (z.B. integer initial stets 0)
- Zwang zum expliziten Casting (keine impliziten Typkonvertierungen)
- Beschränkung der Schachtelungstiefe von Blöcken
- Beschränkung der Anzahl von Anweisungen / Zeile
- Beschränkung der Anzahl von Operatoren / Ausdruck

+ Weitere allgemeine Regelungen

- Einrichtung zentraler Resource-Files für
  - Texte der Benutzerführung
  - Hilfetexte
  - Fehlermeldungen
  - Gestaltung der Applikation (Fonts, Farben, ...)
- Hinweise zur defensiven Programmierung
  - if (fopen(...)) ... else ...
- Konventionen für Files
  - Namensgebung für Source-Files, Header-Files, ...
  - Zugriffsberechtigungen auf Files
- Bezeichnung von Varianten (Versionskontrolle)
  - Namensgebung bei mehreren verfolgten Varianten
  - Freigabe von Modulen zum Test

7.3 Anforderungen an ein Programm

Wie bereits festgestellt *oder intuitiv bekannt* gibt es unterschiedlich gute Algorithmen bzw.Programme. Aber, wann ist ein Programm gut oder schlecht? Für die Bewertung gibt es verschiedene Qualitätsmaßstäbe:

1. **Effektivität** (Korrektheit)
2. **Adaptierbarkeit**
3. **Portabilität**
4. **Kompatibilität**
5. **Zuverlässigkeit**
6. **Robustheit**
7. **Verfügbarkeit**
8. **Benutzerfreundlichkeit**
9. **Wartbarkeit**
10. **Effizienz und Leistung**

Ein gutes Programm kann nicht gleichzeitig alle Kriterien erfüllen, manche schließen sich gegenseitig aus oder stehen sogar im Widerspruch zueinander.

- Minimiert man den Speicherverbrauch, so erhöht sich i.d.R. die Rechenzeit und umgekehrt.
  Beispiel: Daten müssen (de-) komprimiert werden vor/nach der Verarbeitung

- Eine Erhöhung der Verfügbarkeit und Benutzerfreundlichkeit führt zu weniger effizienten Programmen.
  Beispiel: Benutzereingaben überprüfen, redundante Datenhaltung,...

Man unterscheidet daher zwischen **absoluten** und **relativen** Anforderungen. Absolute Anforderungen sind beispielsweise Korrektheit und Wartbarkeit.

+ Auf sie kann keinesfalls verzichtet werden!

Alle übrigen haben relativen Charakter und müssen gewichtet werden entsprechend

- dem Einsatzbereich des Programms und
- den gegebenen Randbedingungen in Form von
  - technischen,
  - zeitlichen oder
  - finanziellen Vorgaben.

Typische Beispiele:

| Soll eine zu automatisierende Anlage binnen sehr kurzer Zeit in Betrieb genommen werden, dann leiden sicherlich die **Zuverlässigkeit**, **Benutzerfreundlichkeit** und **Robustheit** der SW.

| Sollen mit einem Produkt neue Wege beschritten werden (z.B. bessere Kommunikationsprotokolle, geeignetere Datenaustauschformate, ...), um bessere Ergebnisse zu erhalten oder komplexere Aufgaben lösen zu können, dann ist man wahrscheinlich nicht mehr **kompatibel** zu früheren Programmen.

| Werden die Kosten von redundanten Mehrrechnersystemen nicht akzeptiert, dann ist erhöhte **Verfügbarkeit** und **Fehlertoleranz** kaum zu erreichen (Beim Space-Shuttle beispielsweise 3 aus 5).

Die möglichen Qualitäten von Software müssen dem Informatiker bekannt sein und sollten mit dem Auftraggeber besprochen werden, unter Hinweis auf die Auswirkungen und Kosten.

\+ Wichtige Anforderungen, welche erfüllt werden müssen oder
auf welche der Kunde (nicht zuletzt aus Kostengründen) verzichtet,
sollten im Pflichtenheft dokumentiert werden.

7.3.1 Korrektheit

Ein Programm arbeitet **korrekt** oder effektiv, wenn es seine **Aufgabe genau erfüllt.** Um dies überprüfen zu können, benötigt man eine **exakte Formulierung** der Aufgabe**,** wie dies bei mathematischen oder logischen Problemen i.a. möglich ist.

Das, was ein Programm tun soll, wird in einer **Spezifikation** beschrieben. Eine exakte Formulierung der Aufgabenstellung eines Programms und damit seine Spezifikation ist in der Praxis meist recht schwierig und deshalb nicht vorhanden.

Man unterscheidet:

- **Externe Spezifikation (Pflichtenheft)**
  - gewöhnlich informelle, verbale Spezifikation
  - in der Regel ein Katalog von Anforderungen an das Programm
  - Eingaben des Benutzers und die Reaktion des Rechners darauf
  - Ausnahmesituationen
  - Zeit- und Leistungsaussagen

- **Interne Spezifikation (Lastenheft)**
  - möglichst formale Beschreibung der Ein-/Ausgabebeziehungen
  - in Form von
    - prädikatenlogischen Aussagen
    - mathematischen Beziehungen
    - abstrakten Datentypen

Aus der externen Spezifikation versucht man die interne Spezifikation, die eine formale Natur hat, abzuleiten. Für eine interne Spezifikation kann eine funktionale Korrektheit definiert werden.

Liegt eine formale Spezifikation vor, so läßt sich - zumindest theoretisch und mit ziemlichem Aufwand - ein Korrektheitsbeweis durchführen.

Leider ist dies nur selten möglich:

- *Was ist die Spezifikation eines verteilten Systems (Teilnehmersystem)?*
- *Was ist die Spezifikation eines Betriebssystems?*
- *Wann sind sie korrekt ?*

Ist aufgrund einer ungenauen Aufgabenstellung (externe Spezifikation) keine genaue interne Spezifikation ableitbar, so ist auch Korrektheit im strengen Sinne nicht nachprüfbar! Dann muß

wenigstens die Erfüllung der externen Spezifikation

(also des Pflichtenheftes) verlangt werden.

Weitere Probleme:

- Übereinstimmung zwischen externer und interner Spezifikation?
- Interne Spezifikation ist statisch, der Betrieb dynamisch:
  - Funktioniert es bei Belastung auch (z.B. plötzlich viele Daten)?
  - Zusammenspiel mit dem Grundsystem (z.B. BS mit 20 User)?
  - Rechtzeitigkeit (in der Prozeßdatenverarbeitung)?
- Funktionale Korrektheit ist nur für gültige Eingaben definiert.
  - Was passiert, wenn Fehlbedienung?

    (Vgl. Zuverlässigkeit, Robustheit, ...)

7.3.2 Adaptierbarkeit

Ein Programm heißt

adaptierbar an eine veränderte Aufgabenstellung

wenn es mit vergleichsweise geringem Aufwand so modifiziert werden kann, daß es die neue Aufgabenstellung erfüllt.

Adaptierbarkeit ist von Portabilität *(Anpaßbarkeit an neues Grundsystem)* zu unterscheiden!

Günstige Voraussetzungen für die Adaptierbarkeit von Software sind:

- modularer Aufbau der Software,
- funktionale, hierarchische Gliederung.

Damit wird gleichzeitig ein hohes Maß an Wiederverwendbarkeit erreicht.

. Wiederverwendbare Softwarebausteine verfügen i.a. über ein hohes Maß an Adaptierbarkeit, insbesondere bei OOP durch Klassen und Vererbung.

. Effizienzsteigernde Maßnahmen verringern i.a. den Spielraum, welcher für eine leichte Anpassung an veränderte Aufgabenstellung notwendig ist.

7.3.3 Portabilität

Programme bauen im Allgemeinen auf einem vorhandenen Grundsystem auf (z.B. Betriebssystem, Runtime-System, BIOS, ...).

Ein Programm heißt portabel, wenn es im Verhältnis zu den Herstellungskosten mit **geringem Aufwand auf andere Grundsysteme** verpflanzt werden kann.

Adaptierbare Software ist in aller Regel auch portabel, aber die zwei Eigenschaften sind auseinander zu halten.

Der Wert der Portabilität wird unterschiedlich eingeschätzt:

- Rechnerhersteller haben nach außen kaum Interesse an portabler Software, um Kunden an sich zu binden.

- Softwarehäuser und die Hersteller intern halten sie besonders wichtig, um Produkte auf vielen Plattformen anbieten zu können.

- Anwender haben großes Interesse, um Entscheidungen bezüglich der Plattform *möglichst frei* treffen zu können.

Unter dem Druck der Anwender geht daher der Trend in Richtung

> | **offener Systeme und**
> | **allgemeiner Standards.**

Beispiele:

Betriebssysteme:	Unix
grafische Oberflächen:	X-Window, OSF-Motif,
Programmierung:	X-Open Portability Guide (XPG),
	System V Interface Definition (SVID)
Netzwerke und Protokolle:	OSI (praktisch: TCP/IP)
Text- und Grafikformate:	Postscript (Fa. Adobe), Display-Postscript
	TeX

Entscheidend dabei ist, daß die Schnittstellen **offen gelegt** werden!

7.3.4 Kompatibilität

Kompatibilität ist die Verträglichkeit zwischen verschiedenen Teilsystemen. Sie liegt vor wenn:

die **Teilsysteme kombiniert verwendet** werden können.

Beispiele:

- Verschiedene Compiler (Pascal, C, Fortran,...) haben das gleiche Object-Format (d.h.Linker kann alle Module zusammenbinden).
- CAD-Programme haben gleiches File-Format.
- Maus-Treiber ist kompatibel mit Druckertreiber (sie tun sich nichts).

Man spricht aber auch von Kompatibilität **bzgl. Softwareergonomie:**

1) Im Bereich Grafical User Interfaces (GUIs regelt eine Gestaltungsanleitung (**Style-Guide**) das Aussehen einer Motif-Applikation, damit hat man
- prinzipiell ähnliche Benutzerführung
- gleich strukturierte Kommandos und (Unter-) Menüs

Datei	**Bearbeiten**	...

Neu
Öffnen
Speichern
...
Beenden

2) Dialog zum Ausdrucken von Files:

Möchten Sie sofort drucken oder im Hintergrund (j/n): **j**
Auf welcher Schnittstelle möchten Sie drucken ?
1 COM1:
2 COM2:
3 LPT1:
4 in ein File
Bitte Nr. (1 bis 4) eingeben: **3**
Wieviele Exemplare sollen gedruckt werden?
Bitte Anzahl (1 bis ...) eingeben: **1**

7.3.5 Zuverlässigkeit

Zuverlässigkeit ist ein statistisches Maß für das Verhalten eines Softwareprodukts unter einer gewissen Last (Auftragsspektrum). Ein System heißt zuverlässig, wenn gemittelt über ein Lastspektrum eine

hohe Wahrscheinlichkeit für die befriedigende Ausführung

von Aufträgen besteht.

Ein Auftragsspektrum ist eine Gewichtung der Aufträge nach bestimmten Erfordernissen, z.B.:

\+ Häufigkeit der Verwendung eines Auftrags
Selten verwendete Funktionen erhalten geringes Gewicht.

\+ Auswirkung bei einer fehlerhaften Ausführung des Auftrags
Lebensnotwendige Funktionen müssen 'totsicher' sein.

Es werden alle Eigenschaften zusammengefaßt, welche die Fehlersicherheit des Programms beeinflussen. Dazu gehören:

- Sicherheit gegen interne Fehler im Programm. ∏ **Korrektheit**
- Nicht jede Störung darf zum Gesamtausfall führen. ∏ **Ausfallsicherheit**
- Auftretende Fehler kompensieren, nützliche Redundanz. ∏ **Fehlertoleranz**
- Bedienfehler darf nicht zum Gesamtausfall führen. ∏ **Robustheit**

Die Zuverlässigkeit setzt sich also zusammen aus den Faktoren Korrektheit, Ausfallsicherheit, Fehlertoleranz und Robustheit.

Man beachte den Unterschied zwischen **Zuverlässigkeit** und **Korrektheit:** Ein System kann zuverlässig sein, obwohl es Fehler aufweist. Voraussetzung ist nur, daß

- die Fehler das System nicht unbenutzbar machen,
- die Fehler nur selten auftreten,
- die Fehler nicht in Fällen auftreten, in denen das System dringend gebraucht wird, bzw. eben unter keinen Umständen ausfallen darf.

Umgekehrt kann ein korrektes System unzuverlässig sein. Dies ist z.B. dann der Fall, wenn

- die Spezifikation der Aufgabenstellung nicht vollständig bzw. widersprüchlich ist,
- durch schlechte Bedienerführung häufig Fehleingaben vorkommen, vielleicht sogar Fehleingaben unbeabsichtigt provoziert werden.

. Benutzer machen in der Bedienung erfahrungsgemäß Fehler, darum müssen Programme gegenüber fehlerhaften Eingaben robust sein.

Zuverlässigkeit ist besonders wichtig bei **Systemen mit Sicherheitsverantwortung.**

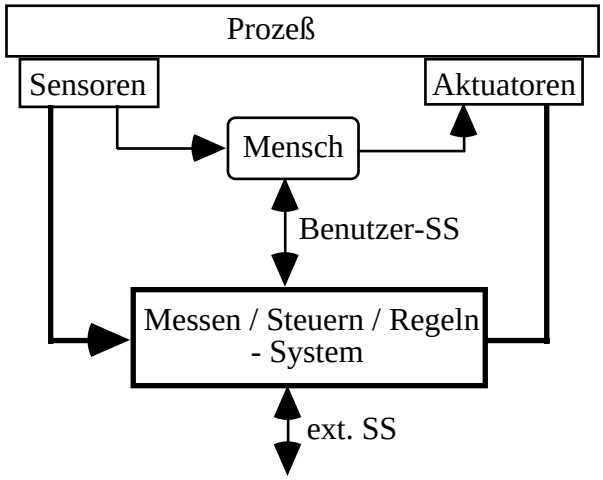
Wesentliche Hinweise / Normen ∏ DIN V 19250, DIN V VDE 0801/01.90.

Sicherheitstechnische Anforderungen werden durch eine **Risikoanalyse des Gesamtsystems** ermittelt. Grundprobleme dabei sind:

- Es müssen Aussagen über sehr kleine Wahrscheinlichkeiten gemacht werden.

- Auch Risiken, welche sich eigentlich einer numerischen (sogar kostenmäßigen) Bewertung entziehen, müssen vergleichbar gemacht werden.

Ein Gesamtsystem hat in der Regel folgenden Aufbau:



7.3.6 Robustheit

Die Robustheit ist eine graduelle Aussage darüber,

wie viele falsche Eingaben vom Programm erkannt und gemeldet werden.

Zur Vermeidung der Auswirkungen von Fehlern müssen diese zuerst

- **entdeckt** (error detection) und dann
- **behandelt** (error recovery) werden.

Da das Programm oftmals die Intention einer Eingabe nicht kennt, muß es:

\+ dem Benutzer den Fehler (genauer das Fehlersymptom)
samt aller potentiell möglicher Diagnosen mitteilen.

> Bei einer automatischen *Error-Recovery* versucht man das Programm soweit in Ordnung (in einen Zustand) zu bringen, daß weitere Eingaben möglich sind und die Auswirkungen auf die internen Daten beseitigt (d.h. rückgängig gemacht) werden.

Beipiel Compiler: der Fehlerfall ist (beinahe) der Normalfall.

Hinweise:

- Die Robustheit einzelner Module ist bei der Integration und beim Test sehr hilfreich.
- Robustheit kann sehr effizienzhemmend wirken, insbesondere wenn jedes Modul die gleichen Daten überprüft. Als Abhilfe bietet sich an, mehrere Versionen eines Moduls mit abgestufter Robustheit zu schreiben.

7.3.7 Verfügbarkeit

Die Verfügbarkeit ist sehr eng mit der Zuverlässigkeit eines Programms verknüpft. Ein Programm besitzt einen hohen Grad an Verfügbarkeit, wenn

Ausfälle nur selten vorkommen und
Standzeiten nur von kurzer Dauer sind.

In vielen Bereichen, insbesondere wenn Prozesse gesteuert, geregelt oder überwacht werden müssen, ist eine hohe Verfügbarkeit gefordert:

lebenserhaltende medizinische Geräte,
Kraftwerke (auch Kern-) müssen ununterbrochen geregelt werden zwar langsam, jedoch bei Störungen evtl. sehr rasche Reaktion,
Verkehrsleitsysteme (Ampeln, Schranken,...),
Flugsicherungsrechner,
Vermittlungsrechner der Post (Ausfallzeit 1h/40Jahre),
Fertigungsstraßen (Stillstand kostet sehr viel Geld).

Im Bereich von betriebswirtschaftlichen Anwendungen sind die Anforderungen i.a. nicht so hoch, jedoch:

Bei Datenbestände von Versandhäusern (z.B.: aktuelle Bestellungen, Lieferungen, Ratenzahlungen, ...) kann eine sehr hohe Verfügbarkeit gewünscht sein.

Weitere Beispiele:

Benötigt man nach einem Systemausfall Stunden, um die Datenbestände in einen konsistenten Zustand zu bringen, so ist die Verfügbarkeit gering.

Fällt ein Rechner nur selten aus, doch müssen die Ersatzteile aus Übersee eingeflogen werden, so ist die Verfügbarkeit gering.

Muß man bei einem PC hin und wieder mal den RESET-Knopf bemühen, das System ist dann aber sofort wieder einsatzbereit, so ist die Verfügbarkeit doch relativ groß.

7.3.8 Benutzerfreundlichkeit

Benutzerfreundlichkeit ist weniger eine Forderung an die Software allgemein, als an die Sollanalyse und die damit verbundene Gestaltung der

Schnittstelle: **Benutzer ⇔ Programm.**

Dazu gehört nicht nur die Definition von Kommandos, Menüs, ... sondern auch der Fehlermeldungen und frühzeitigen Warnungen!

Arbeiten mit einem System unterschiedlich qualifizierte Benutzer, dann sollten eventuell verschiedene Ebenen der Benutzerführung (Hilfe-Funktion) vorgesehen werden. Denn:

\+ Weitschweifige, elementare Fehlermeldungen sind für Profis langweilig und störend.
Zu komplizierte Fehlermeldungen sind für Nicht-Profis wertlos und frustrierend.

Die Stufen sollten aufeinander aufbauen, d.h.:

- Begriffe konsistent in allen Ebenen verwenden.
- Jede Ebene ist vollständig, jedoch unterschiedlich erschöpfend.
- Die Ebenen sind konsistent, oben Gesagtes ist unten noch gültig.

Weitere Kriterien bei der Benutzerfreundlichkeit sind:

- Problemangemessenheit: System soll vom Benutzer nur die Kenntnisse verlangen, die er zur Durchführung der Teilaufgabe benötigt.

- Dialogflexibilität: Ein Benutzer soll ein System entsprechend seinem Kenntnisstand und seiner Übung unterschiedlich benutzen können (Menü, Kommandosprache, Tastenkombination, *„Gedankenübertragung"*).

- Selbsterklärungsfähigkeit: Die Dialogführung sollte dem Benutzer zu jedem

Zeitpunkt anzeigen, in welchem Unterpunkt/Menü er sich gerade befindet, oder in welchem Zustand sich das System gerade befindet:
- wartet auf Eingabe
- bearbeitet noch den zurückliegenden Auftrag
- liest File ein ...

- Fehlertoleranz: Dazu gehören beispielsweise:
  - Falsche Eingaben sollten mit einer Erklärung zurückgewiesen werden.
  - Eingaben sollten rückgängig gemacht werden können.
  - Warnungen vor Befehlen, welche nicht rückgängig gemacht werden können.
  - Aktive Aufforderung zu zyklischem Sichern / Backup von Dokumenten.

Ein Zweig der Informatik, welcher sich mit der optimalen Anpassung von Schnittstellen an die menschlichen Bedürfnisse beschäftigt ist die **Hardware-** und **Software-Ergonomie.**

Hardware-Ergonomie

beschäftigt sich mit der Anpassung der Arbeitsgeräte an die körperlichen und psychologischen Eigenschaften des *"Standard-Hackers":*

- Bildschirmtechnik ∏ MPR-II Norm bzgl. Störstrahlung
- Tastaturtechnik ∏ zweigeteilte Tastatur, getrennter Ziffernblock
- Zeigegeräte für Rechts- u. Linkshänder
- Arbeitsumgebung, DIN 66234, Teil 8: Bildschirmarbeitsplätze
  - Tisch- und Stuhlgestaltung für ermüdungsfreies Arbeiten
  - Arbeitshöhe
  - Lichteinfall, Reflektionen auf dem Bildschirm
  - optimale Arbeitsplatzausleuchtung

Software-Ergonomie

entwickelt Kriterien und Methoden zur Gestaltung interaktiver Programmsysteme, welche den Bedürfnissen der Benutzer nach ausgeglichener Verteilung der

- geistigen,
- körperlichen und
- sozialen Belastung

weitgehend entgegenkommt.

7.3.9 Wartbarkeit

Die Forderung nach Wartbarkeit von Progammen ergibt sich aus der praktischen Erfahrung:

Programme sind fehlerhaft und müssen ein- oder mehrmals angepaßt oder erweitert werden.

Fehlerbeseitigung und Programmveränderung sind wesentliche Komponenten im Lebenszyklus von Software (*40-70% von Aufwand und Kosten*).

Große (und damit teuere) Programme haben i.d.R. eine längere Lebensdauer (um rentabel zu sein) und müssen ggf. mehrmals an sich ändernde:

- Grundsysteme,
- Benutzerschnittstellen,
- Kommunikationsschnittstellen oder
- Benutzerwünsche

angepaßt werden. Eine Neuentwicklung ist dabei wirtschaftlich (noch) nicht vertretbar!

Um Probleme mit dem Kunden zu vermeiden, muß ein Änderungswunsch i.a. auch dann durchgeführt werden, wenn das Produkt nur für kurze Zeit eingesetzt oder eine *Übergangslösung* darstellen sollte.

Ein Programm ist gut wartbar, wenn es **modular aufgebaut** und **gut dokumentiert** ist. D.h., nicht dokumentierte Programme sind nicht wartbar!

Konsequenz von nicht wartbaren Programmen: ist der Mitarbeiter unabkömmlich in einem anderen Projekt oder nicht mehr verfügbar, dann riskiert man erhebliche Mehrkosten, hat also eine falsche Projektkalkulation. Auf jeden Fall muß zusätzliche Zeit für Wiedereinarbeitung investiert werden!

7.3.10 Effizienz und Leistung

Effizienz ist die bestmögliche Ausnutzung aller Ressourcen zur Erfüllung einer Aufgabe. Bei der Softwareentwicklung bedeutet dies die **bestmögliche Nutzung aller Betriebsmittel**.

Die stets benötigten, wichtigsten Betriebsmittel sind:

- der benötigte Hauptspeicher (Speichereffizienz)
- und die benötigte Rechenzeit (Zeiteffizienz).

Die entsprechenden Gütekriterien sind im Gegensatz zu den vorherigen zahlenmäßig erfaßbar und heißen **Speicherverbrauch** sowie **Mittlere Asymptotische Rechenzeit**.

Die Komplexität eines Algorithmus wird durch dessen Speicherverbrauch (sog. **Speicherkomplexität**) und asymptotische Rechenzeit (allg. **Komplexität**) definiert.

Die für die Lösung eines Problems benötigte Rechenzeit und der benötigte Speicherplatz (für Programm und Daten) hängen ab von

- der Art (**Klasse**) des Problems,
- des zur Lösung verwendeten Verfahrens (**Algorithmus**),
- der **Implementierung** und vom
- verwendeten **Grundsystem.**

Als Beispiel betrachten wir die folgenden beiden Algorithmen zur Berechnung der Potenz a^k

Gegeben: a: real; k: integer und k >=0
Gesucht: Potenz p= a^k , p : real (= Ausgabe)

Algorithmus A

```
p= 1;
while (k <> 0) {
   p= p * a;

   k= k - 1;
   }
```

Algorithmus B

```
p=1; q= a;
while (k <> 0)  {
      if (k mod 2) = 1 then p= p * q;
      q= q * q;
      k= k div 2;
      }
```

Der Algorithmus B ist offensichtlich größer als A und benötigt darum auch mehr **Programmspeicher** im Rechner.

Beide Algorithmen kommen mit einer unterschiedlichen Menge an **Datenspeicher** aus.

Datenspeicher:	A	B
real-Zahlen	a,p	a,p,q
integer-Zahlen	k	k
Summe:	2 real+1 int	3 real+1 int

Die Anzahl der Rechenschritte (Laufzeit) zur Berechnung der Lösung unterscheidet sich ganz erheblich bei beiden Versionen und ist jeweils eine Funktion der Eingabegröße k.

Algorithmus A ist klar, daher sei nochmals kurz die Lösungsidee von Algorithmus B skizziert:

k= | k_n | | k_1 | k_0 | z.B. 1021= 11.1111.1101

$a^k = k_n a^k * ... * k_1 a^2 * k_0 a^1$

- innerhalb der Schleife wird jeweils das k_i (durch k mod 2) berechnet (ist also gleich 0 oder 1),
- q wird (durch die Anweisung q= q * q) zu a, a^2, a^4, ...
- bei jedem Durchlauf wird p genau dann mit q multipliziert, wenn ein k_i = 1 ist.

Somit erhalten wir folgende Ergebnisse:

	A		**B**	
	Max	Mittel	Max	Mittel
Vergleich	k	k	$2 \log_2 k$	$2 \log_2 k$
Add/Sub	k+1	k+1	0	0
Mult/Div	k	k	$3 \log_2 k$	$2.5 \log_2 k$
Zuweis.	2k+1	2k+1	$3 \log_2 k+2$	$2.5 \log_2 k+2$
Summe	5k+2	5k+2	$8 \log_2 k +2$	$7 \log_2 k +2$

Bei Berechnung der Summe wurde angenommen, daß die obigen elementaren Operationen alle einen Aufwand von 1 erfordern (eine Maschinenoperation).Diese Annahme gilt allerdings nicht uneingeschränkt, denn Addition und Multiplikation von Integer bzw. Float kann ganz unterschiedliche Anzahl von Zyklen benötigen!

Da exakte Größen nur von untergeordneter Bedeutung sind, beschränkt man sich bei der Abschätzung des Aufwands (obere Grenze) auf den **"Groß-O"-Kalkül** :

Ist die Laufzeit l(k) für alle Eingaben der Größe/Länge k bis auf einen konstanten Faktor C_1 und einen Absolutbetrag C_0 gleich, so spricht man von einer

linearen Laufzeit	$l(k) = C_1 * k + C_0 =$	**$O(k)$.**

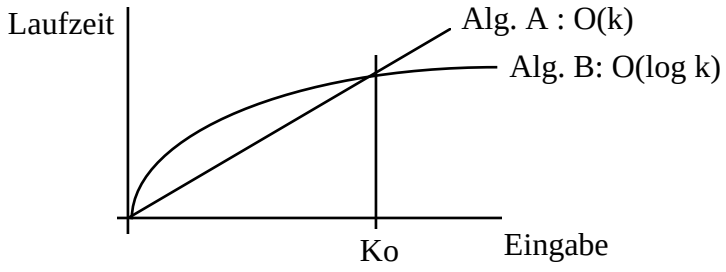
Entsprechend spricht man von einer

quadratischen Laufzeit bei	$l(k) = C_2 * k^2 + ... =$	**$O(k^2)$**
polynomiellen Laufzeit bei (der Ordnung n)	$l(k)= c_n * k^n + ... + c_0 =$	**$O(k^n)$**
exponentiellen Laufzeit bei (p höchstens Polynom!)	$l(k)= c * 2^{p(k)} =$	**$O(2^{p(k)})$**
logarithmischen Laufzeit bei	$l(k)= c * \log(k) =$	**$O(\log(k))$**

In diesem Sinn hat

- A eine lineare Laufzeit O(k) und
- B nur eine logarithmische Laufzeit O(log k).

D.h., selbst wenn Algorithmus A bei kleinen k schneller arbeitet als B, so gibt es doch ein k_0 derart, daß für alle $k > k_0$ B stets schneller ist als A.



7.4 Programmiersprachen

Die Wahl einer Implementierungssprache hat wesentlichen Einfluß auf:

- Programmlesbarkeit (-transparenz, Selbsterklärungsfähigkeit)

- Programmiergeschwindigkeit (Lines of Code / h)

- Programmumfang

- sowie auf die Qualitäten der Software, nämlich:

  Effektivität, Zuverlässigkeit, Robustheit,
  Adaptierbarkeit, Portabilität, Kompatibilität,
  Benutzerfreundlichkeit,
  Wartbarkeit,
  Effizienz.

Es gibt eine prinzipielle Einteilung der Programmiersprachen in:

• Imperative Programmiersprachen
  - Assemblersprachen
  - klassische Hochsprachen
  - objektorientierte Sprachen

• Deklarative Sprachen
  - wissens-orientierten Sprachen
  - regelorientierte Sprachen
  - frame-orientierte Sprachen

Eine andere Einteilung spiegelt mehr die historische Entwicklung wieder: man teilt Programmiersprachen in sogenannte Generationen ein.

1. Generation ⎩ **Maschinensprachen**

- entspricht gerade der Menge gültiger Maschineninstruktionen (binär) einer speziellen CPU,
  z.B. Binär-Code von intel-8080, PDP11

2. Generation ⎩ **Assembler-Sprachen**

- mnemotechnische Abkürzungen für Maschineninstruktionen
  faßt oft die Befehle einer Prozessorfamilie zusammen
  z.B. Intel 80x86-Assembler, VAX-Assembler, 680x0-Assembler

3. Generation ⎩ **Höhere Programmiersprachen** (HPS)

- anwendungsspezifische HPS
  - SIMULA
  - PLM zur Programmierung von Mikroprozessor-Applikationen
  - COBOL im Bereich Wirtschaftswissenschaften
  - RPG II, ...
- universelle HPS
  - Algol, Basic, Fortran, Pascal, Modula, C, Ada
- objektorientierte HPS
  - Smalltalk, Oberon, Eiffel, Objective-C, C++ , Java

4. Generation ⎩ **Anwendersprachen**

- Datenbanksprachen
  - DDL
  - SQL
  - NATURAL 2
- Sprachen spezieller Applikationen
  - Skript-Sprachen
  - PPS
  - dBASE
  - OPEN ACCESS
  - EXCEL

⎪ Es sind deklarative Sprachen (d.h. Anwender muß nicht den Algorithmus angeben, sondern nur das gewünschte Ergebnis beschreiben).

⎪ "Programmieren" besteht aus dem Zusammenfügen von *Bausteinen*, welche spezielle Teillösungen bereitstellen.

Dazu ein Beispiel in NATURAL 2

```
find KUNDE
        with PLZ=76000
        and UMSATZ > 50000
sorted by NAME
display NAME  PLZ  ORT  UMSATZ
at end of data
        write 'Summe Umsatz: ' sum(UMSATZ)
```

- Oft eigene Entwicklungsumgebung für speziellen Problembereich
- Im Anwendungsbereich schnellere Programmierung, weniger Fehler
- Weniger Flexibilität als bei 3 GL
- Keine Optimierungsmöglichkeiten

5. Generation \ Sprachen der **Künstlichen Intelligenz** (KI /AI)

- klassische Sprachen
  LISP, PROLOG

- Shell-Sprachen
  KEE, Babylon, ART, Knowledge-Craft, KES, LOOPS, S.1

- Es sind Wissensrepräsentationssprachen auf der Basis von Regeln bzw. Frames.
- Der Programmierer braucht keine Kenntnisse über Ablauflogik, Struktur der Problemlösung.
- Programme sind auf Rechnern mit gleichem Entwicklungssystem portabel.
- Fehleranfälligkeit beschränkt sich auf falsches bzw. inkonsistentes Wissen, Debug oft über spezielle Erklärungskomponente möglich.
- Performance der Applikation oft sehr schlecht.
- Anwendung bei Nicht-Standard-Problemen, schlecht strukturierten Problemen etwa bei Auskunft, Beratung, Diagnose, Konfiguration.

Dazu als Beispiel das *Problem der Türme von Hanoi* in PROLOG

```
hanoi(N):- loesung(N,links,mitte,rechts).
loesung(0,_,_,_).
loesung(N, A, B, C):- M is N-1;
        loesung(M, A, C, B); write( A, "-->", B); loesung(M, C, B, A).
?- hanoi(17).
```

6. Generation(?) \ **Neuronale Netze**

| Es gibt eine Trainings- bzw. Lernphase statt der Programmierung (1-4 GL) oder der Wissensspezifikation (5 GL).

| Es ist nicht mehr nachvollziehbar, wie Lösungen entstehen (Selbstorganisation).

| Evtl. sind die Systeme selbstlernend, und daher nach einer Anfangszeit fehlerfrei???!!!

8 Überprüfung und Test

Die Überprüfung eines Programmes kann erfolgen durch:

- **Programmverifikation**
- **Programmvalidierung**
- **Systematisches Testen.**

Dies sind abgestufte Methoden, welche sich ergänzen! Programmvalidierung und systematisches Testen muß eingesetzt werden, wenn eine Verifikation nicht möglich ist wegen:

- der Größe des Programms,
- mangelnder oder ungenauer Spezifikation oder
- wegen Kosten, Qualifikation des Personals.

8.1 Übersicht über die Verfahren

Für die **Programmverifikation** gilt:

- Programmverifikation setzt die Existenz eines geeigneten **Kalküls zur Durchführung des Korrektheitsbeweises** voraus (z.B. Hoare oder Dijkstra' sche Verifikationsregeln).
- Die Methode macht bereits bei der Verifikation von Programmen praktischer Größe einen so großen Aufwand, daß das Beweisverfahren mechanisiert werden muß, d.h. **automatische Beweiser** werden eingesetzt.
- Automatische Beweiser sind selbst Programme beträchtlicher Größe *(sie sind darum selbst i.d.R. nicht verifiziert?!).*
- Es müssen Programmiersprachen verwendet werden, welche eine automatische Verifikation unterstützen.

In der Praxis spielt die Verifikation noch keine große Rolle.

Unter der **Programmvalidierung** versteht man die Prüfung der Spezifikationstreue einschließlich etwa geforderter Leistungsdaten. Die Prüfung kann beispielsweise bestehen aus:

- Simulation
- Gegenüberstellung von Programm und Testprogramm (evtl Testautomat)
- Plausibilitätsbetrachtungen
- Code-Inspektion
- Messungen.

Systematisches Testen bedeutet das Ausführen des Programms mit einem vorher festgelegten Satz von möglichen Eingabedaten, den sogenannten Testdaten.

+	Tests können praktisch nur die Anwesenheit von Fehlern zeigen, keinesfalls die Fehlerfreiheit.

Bei der Auswahl von Testdaten ist daher besondere Sorgfalt notwendig!

. Testdaten werden nicht erst nach erfolgter Implementierung ausgewählt, sondern sind Bestandteil des Entwurfs! Bereits beim Entwurf sollte ein möglichst **vollständiger** und zugleich **minimaler** Satz von Tests bzw. Testdaten bestimmt werden.

Vollständig: Jede Anweisung des Programms muß mindestens einmal mit Grenzdaten durchlaufen werden.

Minimal: Den gleichen Test mit unterschiedlichen Testdaten für einen Modul braucht man nicht mehrmals durchzuführen (wegen kombinatorischer Explosion und resultierendem Aufwand und "Nachdenken lohnt immer").

8.2 Klassifikation von Fehlern

Fehler lassen sich naiv nach ihrem Typ einteilen:

Formale Fehler wie z.B.:
- Schreibfehler
- falsche Anwendung der Programmiersprache

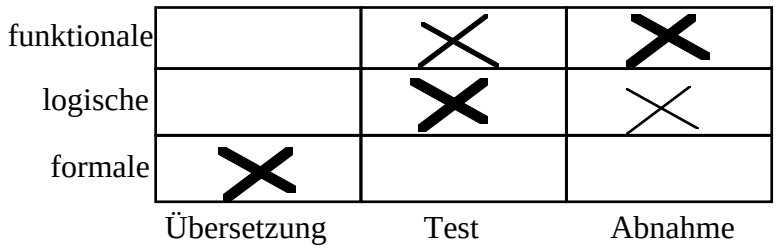
Logische Fehler wie z.B.:
- angewandtes vor definierendem Auftreten von Variablen
- Schleifenbedingung ist keine Funktion des Schleifenrumpfes (außer bei Prozeßdatenverarbeitung!)

+ Ziel ist klar, aber "Denkfehler" bei der Realisierung

Funktionale Fehler wie z.B.:
- Divergenz zwischen Spezifikation und realisierter Funktion

+ Bereits die Annahme über das Ziel ist falsch:
- Ziel wurde falsch / ungenau spezifiziert
- Ziel-Spezifikation wurde vom Programmierer falsch interpretiert.

Entsprechend werden Fehler zu unterschiedlichen Zeitpunkten entdeckt:



Fehler lassen sich auch aus der Sicht eines Übersetzers klassifizieren:

Lexikalische Fehler wie z.B.:
- Zeichen entsprechen nicht dem Alphabet der Sprache (ü, ^ k,...)
- reservierte Wörter falsch geschrieben

Syntaxfehler wie z.B.:
- falsche Programmiersprachenkonstrukte bzw.
- reservierte Wörter an falscher Position im Text
  d.h. Programm läßt sich **nicht** aus Syntaxdiagrammen ableiten

Fehler der statischen Semantik
- Verwendung nicht vereinbarter Objekte (z.B. Variablen)
- Zuweisung ungleicher Datentypen
- angewandtes vor definierendem Auftreten einer Variablen

Fehler der dynamischen Semantik
- Division durch Null
- ungültiger Pointer
- nicht terminierende Schleife / Rekursion

Formale Fehler können von einem Übersetzer sofort erkannt werden, bzw. können durch Hinzufügen von zusätzlichem Programmcode gezielt behandelt werden:

- Auslösung eines Signals
- Aufruf einer vom Anwender definierten Funktion (Error-Handler)

Logische Fehler können im allgemeinen nicht von einem Übersetzer erkannt werden. Ausnahmen sind z.B.:

- nicht erreichbarer Code
- deklarierte und nicht verwendete Variablen, Funktionen.,...
- konstante Bedingungen in Schleifen

Sie werden i.a. als Warnungen an den Benutzer ausgegeben

Andere typische logische Fehler, wie nicht terminierende Schleifen bzw. allgemein ungewollter, d.h. falscher Programmablauf, können prinzipiell nicht im Übersetzer entdeckt werden, da sie stets auch in der Absicht des Programmierers liegen können.

> Logische Fehler werden im allgemeinen erst durch ihr Symptom erkannt, nämlich durch die *Nichteinhaltung der Spezifikation.*

Logische und funktionale Fehler treten häufig in Kombination auf, wobei erst eine Analyse die tatsächliche Ursache zeigen kann. Es gibt mehrere Test- und Überprüfungsmethoden, um logische und funktionale Fehler zu entdecken.

8.3 Programmvalidierung

Validierung heißt:

Die Korrektheit des Programms / Moduls plausibel bzw. einsichtig machen!

Dazu gibt es diverse manuelle und automatische Methoden, die im folgenden vorgestellt werden.

8.3.1 Schreibtischtest

Eine Testperson überprüft am Schreibtisch ein Programm: am Shreibtisch bedeutet ohne Rechner, d.h. die Überprüfung erfolgt an Hand des Listings mit Papier und Bleistift. Der Ablauf ist folgendermaßen:

- Testdaten für die Eingabe auswählen:
  - Testdaten sollen nicht zu kompliziert sein.
  - Testdatensatz aus Entwurfsphase kann verwendet werden.
- Programmablauf anhand des Listings schrittweise durchspielen:
  - Testperson übernimmt Aufgabe eines Interpreters.
  - Aktuelle Variablenbelegungen werden mitgeschrieben.
  - Gleichzeitig wird das Verhalten bei leichter Variation der Eingabe-Daten im Auge behalten, z.B. Grenzfälle bei Verzweigungen.

Es gibt psychologische Probleme, wenn ein Autor seine eigenen Programme testet:

- Die Methode ist langwierig und mühevoll.
- Sie erfordert hohe Konzentration und ist selbst sehr fehleranfällig.
- Oft werden Variablen vergessen und später nachgetragen.
- Sie spiegelt oft nur die eigene Meinung über die Korrektheit des Programmes wieder.

Die Vorteile, wenn ein andere Person den Test durchführt, sind:

- Die Testperson arbeitet sich automatisch in das Programm ein.
- Eine unabhängige Person hat das Programm verstanden und hält es für korrekt.

8.3.2 Walkthrough

Ein Walkthrough ist analog dem Schreibtischtest, jedoch sollten mind. 2 max. 4 Personen daran teilnehmen, darunter auch der Autor. Zu dem Team sollten wahlweise hinzugezogen werden:

- erfahrener Programmierer,
- Experte für die Programmiersprache,
- weiterer Programmierer aus dem Team des Autors,
- eventuell neuer Mitarbeiter (unvoreingenommen, neue Ideen).

Folgende Regeln sollten bei einem Walkthrough beachtet werden:

- Relevante Testfälle nicht zu komplex wählen (sie sollen von Menschen, nicht vom Compiler bearbeitet werden.

- Erfordert Vorbereitung der Teilnehmer, diese sollten Listing kennen!

Auch hier wird wieder der Programmablauf anhand des Listings schrittweise durchgespielt. Dabei sollte jedoch nicht das korrekte Durchspielen der Testfälle im Mittelpunkt stehen, sondern das gedankliche Nachvollziehen von Varianten und die Beantwortung von auftretenden Fragen (Was wäre, wenn i nicht diesen Wert hätte, sondern ...).

Die Vorteil dieses Verfahrens sind:

- weniger fehleranfällig, da mehrere Personen beteiligt sind,
- bessere / schnellere Einarbeitung, weil Autor Teammitglied ist,
- Erfahrungen von Spezialisten werden dabei weitergegeben.

Die Beseitigung von Programmierfehlern ist nicht Bestandteil des Walkthrough sondern wird vom Programmierer gesondert (später) erledigt! Bei entsprechender Zahl von Fehlern ist ein erneuter Termin für einen Walkthrough (mit dem dann berichtigten Programm) zu vereinbaren.

Erfahrungsgemäß ergeben sich beim Ändern von Programmen mehr Fehler, als beim Schreiben neuer (Fehler/Lines-Code)!

8.3.3 Codeinspektion

Codeinspektion bedeutet: Sequentielles Lesen und Verstehen des Programms durch eine Testgruppe bestehend aus 3 bis 4 Mitgliedern:

- Moderator (QS-Fachmann, oder erfahrener Progammierer, er muß keine Details vom Programm kennen. Seine Aufgaben sind: Verteilung von Unterlagen, Leitung der Sitzung, Anfertigung von Protokoll mit Fehlerliste.)
- Autor
- Designer (Entwurf des Moduls)
- Testspezialist

Rechtzeitig vor der Inspektionssitzung müssen Spezifikation und Listing verteilt werden. Die Teilnehmer müssen sich vorher mit dem Stoff auseinandergesetzt haben. Das Vorgehen während der Sitzung:

- Autor erklärt die Programmlogik Anweisung für Anweisung.
- Entstehende Fragen werden vom Team verfolgt und beantwortet. Die Erfahrung zeigt, daß dabei viele Fehler vom Autor selbst (während des Vortragens) erkannt werden.
- Das Programm wird mit Hilfe einer Checkliste häufig auftretender Fehler analysiert.

Die Dauer einer Sitzung beträgt zwischen 90-120 Min. Auch hier gilt:

Die Beseitigung von Fehlern ist nicht Gegenstand der Sitzung!

Werden grobe/weitreichende oder viele Fehler entdeckt, so ist die Sitzung zu einem späteren Termin mit einer berichtigten Version zu wiederholen! Die Liste der entdeckten Fehler dient gleichzeitig der Überarbeitung der Checkliste und damit auch der Verbesserung künftiger Inspektionen.

Weitere Effekte von Walkthrough und Inspektion:

- Programmierer erkennt eigene Fehler und lernt (hoffentlich) daraus.
- Erfahrung / Programmierstil anderer kann übertragen werden.
- Progammdokumentation kann überprüft, berichtigt, verbessert werden.

8.3.4 Liste der häufigsten Fehler

Hier nun noch die Liste der Fragen, die auf die häufigsten Fehler hinführen:

- Ist jede Variable auch vorher definiert?
- Sind Felder und Strings richtig initialisiert?
- Sind die Laufvariablen in geschachtelten Schleifen disjunkt?
- Liegen Indizes innerhalb der Grenzen (1. Element Nr.0 | 1 ?)?
- Sind die Indizes ganzzahlig?
- Gibt es ungültige Zeiger?
- Gibt es Über- / Unterlauf / Rundungsfehler von Variablen?
- Kann Division durch 0 auftreten?
- Wird die Priorität der Operatoren richtig verwendet bzw. geklammert?
- Sind die boolschen Ausdrücke korrekt?
- Werden bei Entscheidungen alle Fälle berücksichtigt?
- Terminiert jede Schleife?

Vgl. [Mye 79]

8.3.5 Automatische Methoden der Programmvalidierung

Wenn Fehler entdeckt werden können, dann i.d.R. durch **komplexe und aufwendige Übersetzer**, welche das Quellprogramm durch Syntax- u. Semantikprüfung so weit wie überhaupt möglich analysieren.

Weitere automatische Analysen beschränken sich darum oftmals auf formale Aspekte wie:

- keine zu langen Zeilen im Programmtext,
- keine zu komplexen arithmetischen oder logischen Ausdrücke,
- Überprüfung der Einrückregeln bei Blöcken Funktionsdeklarationen,
- Einhaltung von Programmierrichtlinien.

Aus den ermittelten Daten (Häufigkeit des Auftretens) werden **Software-Metriken** erstellt. Sie lassen jedoch nur sehr begrenzt Rückschlüsse auf die Qualität von Software wie z.B. Modularität, Wartbarkeit, Testbarkeit, Sicherheit, ... zu.

8.4 Das Testen von Programmen

Eine Möglichkeit der quasiautomatischen Programmvalidierung bietet das bekannte Testen von Programmen. Erschöpfende Tests sind in der Praxis nicht möglich, darum ist die zentrale Frage:

> Welche Untermenge aller denkbaren Testfälle bietet die größte Wahrscheinlichkeit, möglichst viele Fehler zu finden?

Hierzu helfen uns die Methoden des systematischen Testens, bei denen Testdaten über Grenzwertanalyse, Äquivalenzklassenbildung und durch Erfassung aller

möglichen Codesequenzen bestimmt werden. Daneben gibt es auch weniger systematische Methoden wie:

• **Anwendertestfälle**
Vom Personenkreis der künftigen Anwender wird eine Menge von typischen Eingabedatensätzen erstellt, welche möglichst realistischen Ursprungs sind.

• **Testen mit automatisch erzeugten Zufallszahlen**
Einfachste und schwächste Methode, weil Testfälle leicht zu generieren sind, aber die Chance eine optimale Untermenge zu finden sehr gering ist.

• **Fehlererwartung**
Aus der Praxis gewonnene Informationen über besonders fehlerträchtige Situationen (siehe dazu auch die Liste von typischen Fehlern weiter oben):
- Verarbeitung von Wert 0, Leer- Füllzeichen (Blanc, Tab, Newline)
- Länge von String = 0, leerer String
- Probleme bei Datumsarithmetik, Tages-, Monats-, Jahreswechsel
- Überlauf von Zwischenpuffern
- mögliche Rundungsfehler
- Vertauschung der Eingabereihenfolge

8.4.1 Testmethoden

Man unterscheidet im wesentlichen zwischen den sogenannten Blackbox- und Whitebox-Tests.

Kennzeichnend für **Blackbox-Tests** sind:

• interne Realisierung des Moduls unbekannt (und uninteressant)

• lediglich Überprüfung anhand der Schnittstellen-Spezifikation des Moduls

Beispiel: Man entwerfe Testdaten für die Prozedur:
Eingabe: 3 Seitenlängen eines Dreiecks
Ausgabe: ungleichseitig, gleichschenklig, gleichseitig

Einige Testfälle:

1) ungleichseitig; bei 1,2,3 oder 2,5,10 keine Ja Antwort, da kein Dreieck!
2) gleichschenklig; bei 2,2,4 keine Ja Antwort, da kein Dreieck!
3) bei gleichschenklig alle Permutationen mit 2 gleichen Seiten
4) Testfall mit einer Seite gleich 0 oder -4
5) Testfall 0,0,0
6) Testfall mit nicht ganzzahligen Werten
7) Testfall mit nur 2 Eingabewerten

Kennzeichnend für Whitebox-Tests sind:

- interne Realisierung liegt in Form des Quell-Codes vor.

Whitebox-Tests werden manchmal auch als Glasbox-Tests bezeichnet.

Während Testfälle für Blackbox-Tests bereits in der Designphase entworfen werden können und sollten, denn zu diesem Zeitpunkt ist ja die Schnittstelle und das Verhalten bezüglich der Schnittstelle bekannt, handelt es sich bei den Whitebox-Tests um zusätzliche Tests, die nun nachdem die Implementierung bekannt ist, entworfen werden können. Die Definition von neuen zusätzlichen Testfällen ist notwendig, da nun auch die Eigenschaften eines Moduls durch die Implementierung genauer bekannt sind.

Im Folgenden wenden wir uns einer Reihe von Methoden zu, mit deren Hilfe Whitebox-Testdaten gewonnen werden können.

8.4.2 Grenzwertanalyse und Äquivalenzklassenbildung

Ein Grenzwert ist ein Wert am Rand einer Äquivalenzklasse (ÄK). Eine Äquivalenzklasse ist eine Untermenge gleichartiger gültiger bzw. ungültiger Eingaben derart, daß:

> falls eine Eingabe aus einer Klasse von gültigen Eingaben eine gültige Ausgabe erzeugt, so auch jeder andere Wert aus dieser Klasse,

> falls eine Eingabe aus einer Klasse von ungültigen Eingaben einen Fehler erzeugt, so auch jeder andere Wert aus dieser Klasse.

Beispiel:

```
if (( a < 5 ) OR ( a > 10 )) then ....
```

Klasse1:= {-MAXINT,...,3,4,11,12,... MAXINT}
Klasse2:= {5,6,...,10}

Ü Welches sind entsprechende Äquivalenzklassen bei dem Programmstück:

```
if (( a < 5 ) OR ( a > 10 ))
  {  if ( a mod 2 = 0 ) •••
     •••
else
  {  •••
```

8.4.3 Erfassung / Ausführung aller möglichen Codesequenzen

Ein Programm wird zerlegt in:

Maximal gerade Programmstücke
(wird die erste Anweisung ausgeführt, dann auch sicher die letzte!)

Programmverzweigungen (auch Mehrfachverzweigungen)
(abhängig von Bedingungen, welche per & bzw. **|** verknüpft sind)

Daraus ergibt sich ein **Programmflußgraph** mit:

Knoten sind die Programmverzweigungen sowie
ein Start- und (evtl. mehrere) Endknoten (ausgezeichnete Knoten).
Gerichtete Kanten sind die maximal geraden Programmstücke.

Wir unterscheiden:

- **Anweisungen** (C0)
- **Pfade** (C1)
- **Bedingungen** (C2)
- **Pfade und jeder Schleifengrenze** (C3)

Bei einem C_i-Test sind so viele und solche Testdatensätze zu wählen, daß möglichst

C0-Test:
jede Kante mindestens einmal durchlaufen wird (damit auch jede Anweisung),

C1-Test:
jeder mögliche Pfad von Start bis Ende mind. einmal durchlaufen wird,

C2-Test:
alle Teilbedingungen (in einer Programmverzweigung) mindestens einmal mit dem Wert wahr und einmal mit dem Wert falsch durchlaufen werden,

C3-Test:
alle möglichen Pfade von Start bis Ende für jede enthaltene Schleife mit allen charakteristischen Werten mind. einmal durchlaufen werden, wobei charakteristische Werte für eine Schleifenbedingung sind:
true / false
Index < min, Index = min, Index > min,
Index < max, Index = max, Index > max
•••

Die Durchführung eines Whitebox-Tests wird über den Deckungsgrad des Tests z.B. 100% C0 oder 92% C1 angegeben.

Es ist offensichtlich, daß eine 100%-tige Überdeckung i.a. nicht erreicht werden kann. Beispielsweise ist 100% C3 bei geschachtelten Schleifen kaum möglich wegen der kombinatorischen Explosion.

Bei komplexen Bedingungen gilt:

- jeder Teil mindestens einmal true und einmal false und
- ganze Bedingung mindestens einmal true und einmal false.

8.4.4 Beispiel

Man entwerfe einen C2-Test für das folgende Programm:

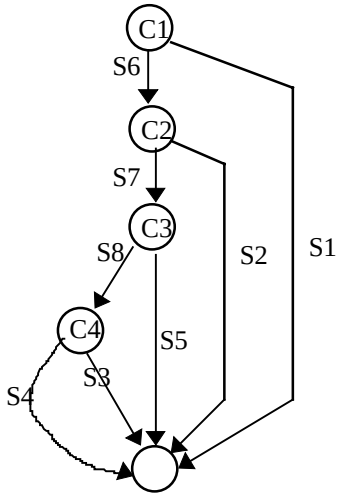
Geg.: int a,b,c
Ges.: Aussage, ob Dreieck mit den Seiten a,b,c gleichschenklig, gleichseitig oder ungleichseitig ist.

Algorithmus:

```
   C1                                          S1
if ( a <= 0) | (b <= 0) | (c <=0)  then  write "Seitenlänge muß > 0 sein !";
   C2                                          S2
if ( a+b <= c) | (a+c <= b) | (b+c <= a)  then  write "kein Dreieck";
   C3
if ( a=b) | (a = c) | (b = c)
        C4                                  S3
  then  if (a = b) & (b = c)  then  write "gleichseitig" ;
                                    S4
                              else  write "gleichschenklig" ;
     S5
  else write "ungleichseitig" ;

end
```

Programmflußgraph:



mögliche Pfade sind:
(S1)
(S6,S2)
(S6,S7,S5)
(S6,S7,S8,S3)
(S6,S7,S8,S4)

In der folgenden Tabelle sind Testfälle mit je einem Testdatensatz aus einer Klasse zu finden, dabei gilt: ci.j entspricht Teilbedingung j von Bedingung ci, und innerhalb von Teilbedingungen entspricht 0=false, 1=true.

C1.1	C1.2	C1.3	C2.1	C2.2	C2.3	C3.1	C3.2	C3.3	C4.1	C4.2	a b c	Ergebnis:
1	0	0									-2 3 3	Seitenl. soll > 0
0	1	0									2 -3 6	Seitenl. soll > 0
0	0	1									1 2 -1	Seitenl. soll > 0
0	0	0	Testfall ist in späteren Testdaten enthalten									
			1	0	0						2 3 5	kein Dreieck
			0	1	0						2 6 3	kein Dreieck
			0	0	1						4 2 1	kein Dreieck
			0	0	0	Testfall ist in späteren Testdaten enthalten						
						1	0	0			3 3 1	gleichschenklig
						0	1	0			4 5 4	gleichschenklig
						0	0	1			2 3 3	gleichschenklig
						0	0	0			3 4 5	ungleichseitig
									1	1	6 6 6	gleichseitig
									0	1	2 3 3	gleichschenklig
									1	0	4 4 3	gleichschenklig

8.4.5 Zusammenfassung

- Testen ist aufwendig und keinesfalls trivial.
- Testdaten werden nicht "weggeworfen", da sie nach Programm-änderungen, -erweiterungen erneut benötigt werden.
- Oftmals werden gerade bei der Fehlerbeseitigung neue Fehler gemacht!
- Stets Testen und Fehlerbeseitigung trennen (falls nicht unvermeidlich).
- Testausgaben stets durch bedingte Übersetzung einfügen und im Quelltext belassen, evtl. versch. Testlevel (Compiler-Schalter) definieren.
- Debug-Unterstützung ist unverzichtbar.
- Tests nicht erst beginnen, wenn alle Module fertig sind, sondern schrittweise testen und integrieren, da sonst Komplexität zu groß.

8.5 Psychologie des Testens

Es handelt sich beim Testen um die

> Überprüfung von bereits konkret erstellten Programmen / Modulen,
>
> nicht um die Überprüfung von abstrakten Algorithmen).

Testen beinhaltet eine Beurteilung des
- Testobjekts (Arbeit des Autors) und damit indirekt eine Bewertung der
- intellektuellen Arbeit des Autors.

Bei der Bewertung von Programmen handelt es sich somit "vermeintlich" um die Bewertung einer intellektuellen Leistungsfähigkeit (des Autors). Aber:

1) Die Bewertung anderer menschlicher Größen (physischer, wie z.B. Hochsprung) ist weit weniger emotional vorbelastet als diese!

2) Auch ein erstklassiger Programmierer wird, wenn er unter starkem Zeitdruck steht, große private Probleme hat, täglich um seinen Arbeitsplatz fürchten muß, kaum gute oder gar fehlerfreie Programme schreiben.

. Notwendig ist daher ein **partnerschaftliches** Verhältnis zwischen Autor und Tester(n) mit der Überzeugung: "Nobody is perfect".

Weiterhin gilt: Testen ist ein destruktiver Vorgang. Denn Testen ist der Prozeß, ein Programm auszuführen, mit der Absicht dabei Fehler zu finden.

. Positive Arbeitsvoraussetzung ist die **sichere Annahme** von Fehlern, falls nicht, besteht nur geringe Motivation zur Arbeit.

Diese Voraussetzung gilt in aller Regel nicht für den Autor, weil er von seiner Arbeit überzeugt ist, d.h. inbesondere:

. Ein Autor ist ein schlechter Tester für seine Programme.

8.6 Programmverifikation

Programmverifikation ist der streng formale Nachweis der Korrektheit eines Programms mit Hilfe eines Logikkalküls

Die Durchführung ist außerordentlich aufwendig und selbst bei kleinen Programmen praktisch nicht mit vernüftigem Aufwand durchführbar.

Beispiele für solche Kalküle sind die von **Hoare** und **Dijkstra** definierten Verifikationsregeln.

8.6.1 Methode der schwächsten Vorbedingung

Dijkstra definiert die **schwächste Vorbedingung** (weakest precondition):

> Gilt vor Ausführung des Statements S die Bedingung **wp(S,N)**,
> so wird S terminieren und es wird danach N erfüllt sein.

Will man beweisen, daß ein Programm (-stück) S den Zustand V in den Zustand N wandelt (transformiert), so ist zu zeigen, dass:

$$V \rightarrow wp(S,N)$$

Die entsprechenden Regeln für **wp** sind:

D1: wp(S,false) = false

D2: Wenn $V \rightarrow N$ dann: $wp(S,V) \rightarrow wp(S,N)$

D3: **(** wp(S,Q) & wp(S,R) **)** = wp(S, Q & R)

D4: **(** wp(S,Q) | wp(S,R) **)** = wp(S, Q | R)

Zur Verifikation eines Programms muß die Semantik der verwendeten Programmiersprache zunächst mit Hilfe der wp(..) ´ s beschrieben werden.

Beispielsweise lauten die Regeln für die Programmiersprache Pascal:

D5: Zuweisung
$wp(x:= t, N) = N_{t \rightarrow x}$

D6: Konkatenation
$wp(S_1; S_2, N) = wp(S_1, wp(S_2, N))$

D7: Entscheidung
wp (if B then S_1 else S_2 fi, N) = $(B \& wp(S_1, N)) \mid (\neg B \& wp(S_2,N))$

D8: Schleife
wp (while B do S od, N) = ($\exists$ i $\in$ N : $H_i(N)$)

wobei $H_0(N) := \neg B \,\&\, N$
$H_{i+1}(N) := B \,\&\, wp(S, H_i(N))$

D.h. es gilt:
H_0: $\neg B \,\&\, N$
H_1: $B \,\&\, wp(S, H_0(N)) = B \,\&\, wp(S, \neg B \,\&\, N)$
H_2: $B \,\&\, wp(S, H_1(N)) = B \,\&\, wp(S, B \,\&\, wp(S, H_0(N))$
$= B \,\&\, wp(S, B \,\&\, wp(S, B \,\&\, wp(S, \neg B \,\&\, N))$

•••

8.6.2 Beispiel

Gesucht ist die schwächste Vorbedingung dafür, dass nach der Ausführung des Programmstücks:

```
if (x ≥ 0)
    then y:=x ;
    else y:=-x
fi
```

$\boxed{y \geq 0}$ gilt

Die folgende Ableitung ist ein Beweis dafür, daß dies stets gilt:

wp(if $x \geq 0$ then y:=x else y:=-x fi, $y \geq 0$) =

mit D7: $[(x \geq 0) \,\&\, wp(y:=x, y \geq 0)] \mid [(x < 0) \,\&\, wp(y:=-x, y \geq 0)]$

mit D5: $[(x \geq 0) \,\&\, (x \geq 0)] \mid [(x < 0) \,\&\, (-x \geq 0)]$

$(x \geq 0) \mid (x < 0)$

true

Dass die Ausgangsbedingung zu true abgeleitet wurde, bedeutet doch, dass dies offensichtlich ohne jede Vorbedingung (also **stets**!) gilt.

9 Phasenübergreifende Tätigkeiten

9.1 Dokumentation

Allgemeine (Grund-) Regeln zur Dokumentation:

+ **Für jedes Projekt, wird eine Dokumentation angefertigt**.

Dies gilt auch für solche Projekte, welche nach der Analysephase nicht weitergeführt werden. Die Gründe dafür sind:

- die bisher ohnehin geleistete Arbeit wird nicht wertlos,
- Kunde gibt vielleicht zu einem späteren Zeitpunkt den Auftrag,
- bei ähnlichem Projekt reduziert sich die Analysephase,
- kann evtl. für andere (honorierte) Vorstudie verwendet werden.

+ **Dokumentation ist nicht Selbstzweck oder lästiges Anhängsel.**

Wer nicht von ihrer Notwendigkeit überzeugt ist oder keine sehr guten Kenntnisse über das zu dokumentierende Programm besitzt, sollte nicht dokumentieren. Es wird sonst nur massenweise nichtssagendes Papier produziert!

+ **Dokumentation dient den Zielen**:

• Sicherung des rechtlichen Rahmens zwischen Projekt-Partnern

genaue Spezifikation der Aufgabe/Anforderungen und der Projekt-Randbedingungen, wie Termine Kosten,... im Pflichtenheft

• Sicherung des Projektfortschritts

sichere Projektverfolgung und Ist- / Soll- Abgleich von Terminen und Resourcen im Projekt-Netzplan

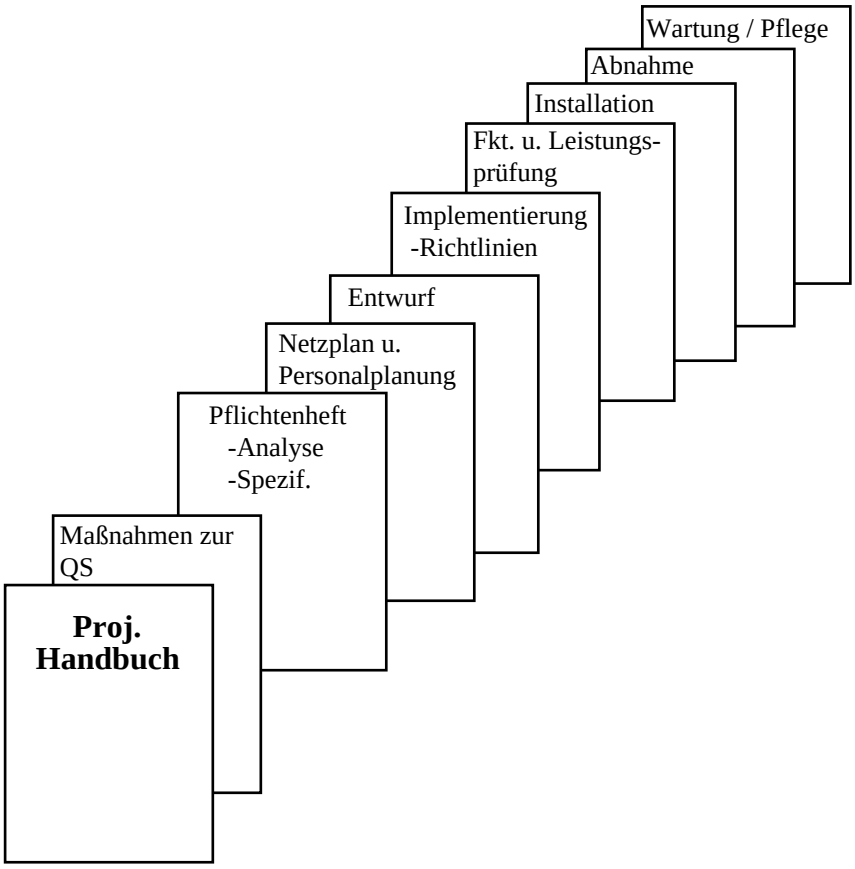
schriftliche Festlegung der Schnittstellen zwischen Projekt-Phasen
SA(/RT)-Diagramme, ER-Diagramme,
Pflichtenheft, Lastenheft,
Structure-Charts, Modulbeschreibungen,
Testergebnisse, Abnahmeprotokoll
Nachvollziehbarbeit von Entwürfen /Entscheidungen / Aktionen
Protokolle von Reviews ("nochmalige Prüfung")

• Qualitätssicherung durch QS-Handbuch (vgl. ISO 9000):

klare Aufgaben- und Kompetenzverteilung auf Personen (personelle Organisation des Projekts) im Organigramm
Zuordnung Aktivitäten zu Personen im Projekt-Netzplan

• Grundlage künftiger Projektplanung

durch Einführung eines allgemeinen (generischen)
Projekthandbuchs



Bei neuem Projekt jeweils:

1. **Kopie des generischen Projekthandbuchs**

2. **fortschreitende Version des PH für akt. Projekt verwalten**

3. evtl. Verbesserungen im genenerischen PH vornehmen (feed back)

4. Projekt-Nachkalkulation
   - zur Korrektur der Maßzahlen für die Kalkulation
   - tatsächlicher Aufwand ⌀ künftige Teminplanung
   - tatsächliche Kosten ⌀ künftige Kostenabschätzung

+ Gute Dokumentation kostet Zeit/Geld, wer nicht/schlecht dokumentiert ist schneller fertig. Die Vorteile sieht nur selten die Person, die sie **mühevoll und gewissenhaft** erstellt hat (psychologisches Problem).

+ Wer sich der gewissenhaften Dokumentation entzieht, möchte gerne weiterhin im "DUNKELN", d.h. unüberprüf- und bewertbar vor sich hin "WURSTELN".

Man unterscheidet:

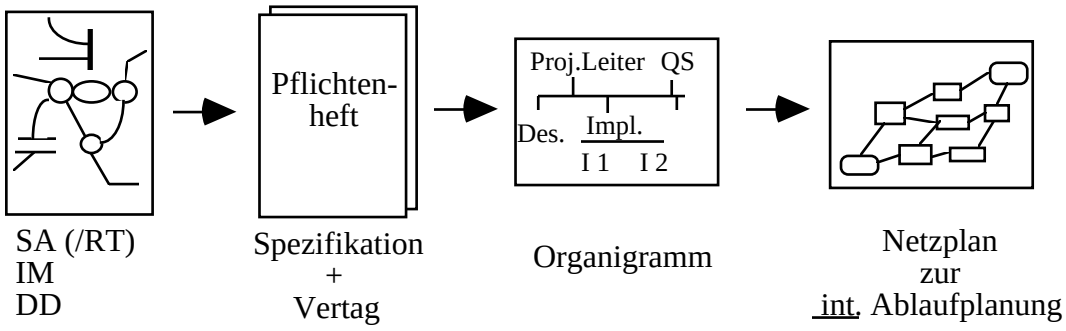
Projektdokumentation	**SW Hersteller**
Programmdokumentation	**SW Hersteller u./o. Kunde**
Benutzerdokumentation	**Kunde**

9.1.1 Projektdokumentation

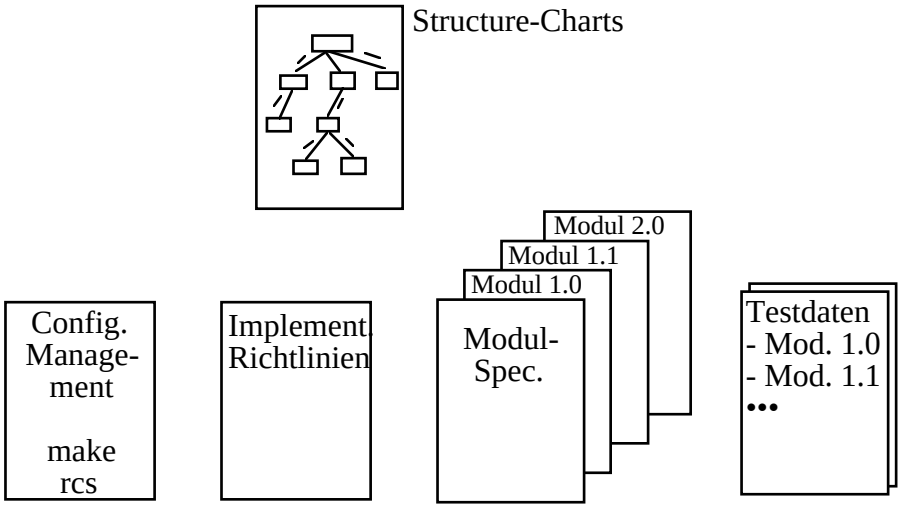
Die Summe aller innerhalb eines Projekts angefertigten Dokumente werden im **Projekthandbuch** zusammengefaßt (i.d.R. elektronisch geführt!).

Im Verlauf eines SW-Projekts läßt sich das Ziel einer jeden Phase durch das Vorliegen der Ergebnisse / Dokumentation überprüfen !

Ergebnisse der **Analyse, Spezifikations- und Organisationsphase** sind:



Ergebnisse der **Entwurfsphase** (Design) sind:



9.1.2 Programmdokumentation

Programmdokumentation, also die Kommentare im Quellprogramm, hat i.a. zwei unterschiedliche Adressaten:

- diejenigen, die das Modul nutzen möchten
  ∏ **Blackbox**-Sicht

- diejenigen, die die Funktionsweise des Moduls verstehen / ändern wollen
  ∏ **Whitebox**-Sicht

Blackbox-Beschreibung ist lediglich:

Genaue Schnittstellen-Beschreibung des Moduls
- Name
- Reihenfolge und -Typ der Parameter

Informationen über evtl. Seiteneffekte
- Zugriff auf globale Datenssrukturen
- Aufruf von extern definierten Funktion
- benötigte Bibliotheksfunktion

Information über benötigte globale Resourcen wie
- allokierter Speicher
- benötigte Files / Geräte / Prozesse

D.h, die Blackbox-Beschreibung ist die Summe dessen, was im **Modulkopf** beschrieben ist, bzw. in der Entwurfsphase andernorts spezifiert wurde (∏ Structure-Charts, Modul-Specs).

Whitebox-Beschreibung gibt erschöpfend Auskunft darüber, **wie** Modul in seinem Inneren funktioniert. Informationen darüber findet man:

+ in der Entwurfsdokumentation (Modul-Spec), soweit dort bereits eine prozedurale Spezifikation des Moduls erfolgt ist. D.h. die Aufgabe des Moduls wurde bereits als "Rezept" spezifiziert, welches angewandt auf die Eingabe- gerade die Ausgabe-Daten liefert, evtl. auch nur grob spezifiziert, z.B.in Form von Pseudo-Code, NS-Diagramm, Programmablaufplan.

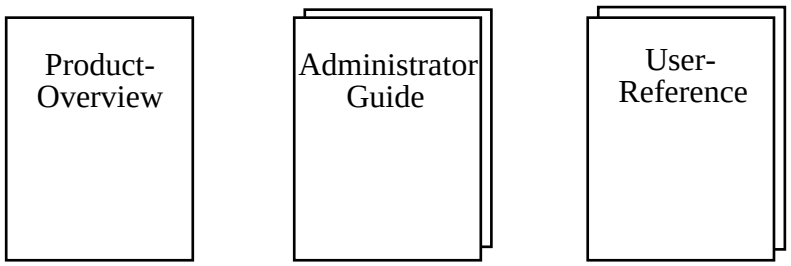
+ in der Quellcode-Dokumentation (Kommentare).

9.1.3 Benutzerdokumentation

Die Benutzerdokumentation ist bei komplexen Systemen neben den ausführbaren Programmen und erstellten Datenfiles das wichtigste Ergebnis eines Software-Projekts.

Daran messen Endanwender i.d.R. auch die Qualität des Gesamtsystems!

Sie umfaßt i.a. die Teildokumente:



Zusätzlich sind eventuell **Vertriebsunterlagen** notwendig, soweit diese nicht von einer eigenen Marketing / Verkaufsabteilung erstellt werden.

Produktbeschreibung (Product-Overview)
Sie soll Antwort auf die Frage geben, ob das SW-Produkt zur Lösung eines gegebenen Problems genutzt werden kann.

Operateurhandbuch (Administrator Guide)
Beschreibt lückenlos alle zum Betrieb des Programmsystems notwendigen einmaligen und stets wiederkehrenden Arbeiten sowie die notwendigen Systemvoraussetzungen zum Betrieb der Software.

Systemvoraussetzungen

- Hardware
  - Rechner / CPU- Typ
  - Hauptspeicher (min. alloc. Speicher)
  - Plattenplatz (min. notwendiger Hintergrundspeicher)
  - notwendige Peripherie

- Software
  - BS-Version und Schnittstellen (z.B. SVID, XPG 4, ...)
  - zusätzlich zu installierende SW (z.B. Shared-Libraries, X11Rx, Tcl/Tk, Datenbank (-server), ...)
  - Hilfsprogramme (z.B. Makroprozessor, Texteditor, Mail, Window-Manager, ...)
  - Kommunikations-Protokolle (z.B. Novell-IPX, TCP/IP, SLIP, ...)

Installationsanweisung (d.h. einmalige Arbeiten ∏ Installation-Guide)

- Einrichtung von reservierten Speicherbereichen auf der Festplatte
- Einrichtung von Zugriffspfaden auf ausführbare Dateien
- Zuteilung von Rechten auf Files, Devices, ...
- Bereitstellung von vorausgesetzter Basis-Software
- Liste mit bekannten Fehlern / Unverträglichkeiten mit anderen Programmen
- Deinstallation

Betriebsanleitung (d.h. regelmäßige Arbeiten ∏ Administrator-Guide)

- Einrichtung (Löschung) neuer Benutzer(-bereiche)
- Tuning, Einstellungen
- Löschen von Arbeitsdateien (Log-, Temp.-Files)
- regelmäßiges Backup
- Fehlerbeseitigung (troubleshooting), Mittler zwischen User u. Hotline
- Update-Service

Benutzerhandbuch (Nachschlagehandbuch ∏ User- / Reference-Manual)

- Einführung in das Werkzeug (Philosophie, Terminologie, erstes Beispiel)
- Funktionsweise von Online-Help
- Ausführliche Beschreibung aller Funktionen (Langfassung)
  (typische Anwendungen, viele Beispiele)
- Kurzbeschreibung der Funktionen mit Verweis zur jeweiligen Langfassung
- Trainings- / Übungsprogramm
- Kompatibilität zu anderen Programmen
- Liste mit Fehlermeldungen und ihre Bedeutung / Behebung
- Stichwortverzeichnis

9.1.4 Konfigurationsmanagement

Mindestanforderung ist "Makefile", welches

- via Versionskontrollsystem die aktuellen (freigegebenen!) Versionen beschafft
- automatisch die Vollständigkeit aller erstellter Module bzw. Bibl. Module überprüft
- entsprechend den Abhängigkeiten neue Object-Module erzeugt bzw. Bibl. Module zusammenbindet
- alle ausführbaren Programme bzw. für deren Ausführung notwendigen Resourcefiles notwendigen Datenfiles, ... erzeugt

9.2 Qualitätssicherung

Die internationale Norm:

ISO9001 Teil 3
Qualitätsmanagement- und Qualitätssicherungsnormen

vom Juni 1993 enthält einen:

Leitfaden für die Anwendung von ISO 9001 auf die Entwicklung, Lieferung und Wartung von Software

Neben

1. Vorwort, Zweck der Norm
2. Verweis auf andere Normen (i.d.R. schwer auf SW übertragbar!)
3. Begriffe (Definitionen)

sind die wichtigsten Kapitel:

*4. Qualitätssicherungssystem - **Rahmen***

4.1 Verantwortung der obersten Leitung

4.1.1 Verantwortung der obersten Leitung des Lieferanten

4.1.1.1 Qualitätspolitik

Die oberste Leitung des Lieferanten muß ihre Politik sowie ihre Zielsetzungen und ihre Verpflichtung zur Qualität festlegen und dokumentieren. Der Lieferant muß sicherstellen, daß diese Politik auf allen Ebenen der Organisation verstanden, verwirklicht und beachtet wird.

4.1.1.2 Organisation

4.1.1.2.1 Verantwortungen und Befugnisse

Die Verantwortungen, Befugnisse und die gegenseitigen Beziehungen aller Mitarbeiter, die leitende, ausführende und überwachende Tätigkeiten ausüben, welche die Qualität beeinflussen, müssen festgelegt werden, insbesondere für jenes Personal, welches die organisatorische Unabhängigkeit und Befugnis besitzen muß um

a) Verhütungsmaßnahmen gegen mögliche Produktionsfehler zu veranlassen;
b) alle Qualitätsprobleme bei Produkten festzustellen und aufzuzeichnen;
c) Problemlösungen nach festgelegten Abläufen zu veranlassen, zu empfehlen oder vorzusehen
d) die Verwirklichung der Problemlösung zu verifizieren;
e) die weitere Behandlung, Auslieferung oder Montage fehlerhafter Produkte so lange zu überwachen, bis die Fehler oder der unbefriedigende Zustand behoben sind

4.1.1.2.2 Mittel und Personal für die Verifizierung

Der Lieferant muß die Forderungen an die interne Verifizierung festlegen, angemessene Mittel bereitstellen und ausgebildetes Personal für die Tätigkeit der Verifizierung bestellen.
Tätigkeiten der Veifizierung müssen Prüfung und laufende Verfolgung der Abläufe und/oder der Ergebnisse von Design, Produktion, Montage und Kundendienst einschließen. Design-Reviews sowie System-, Verfahrens und/oder Produktaudits müssen von Personal ausgeführt werden, das unabhängig von dem ist, welches für die Ausführung der Arbeit direkt verantwortlich ist.

•••

4.2 Qualitätssicherungssystem

•••

4.2.2 Dokumentation des Qualitätssicherungssystems

Alle Elemente, Forderungen und Vorkehrungen für das Qualitätssicherungssystem sollen in einer systematischen und geordneten Weise klar dokumentiert sein

•••

4.2.3 Qualitätssicherungsplan

Der Lieferant sollte auf der Basis des Qualitätssicherungssystems für jede Software-Entwicklung einen Qualitätssicherungsplan erarbeiten und dokumentieren ...

4.3 Interne Qualitätsaudits

•••

4.4 Korrekturmaßnahmen

•••

5. Qualitätssicherungssystem - ***Lebenszyklustätigkeiten***

5.1 Allgemeines

Ein Software-Entwicklungsprojekt sollte nach einem der verschiedenen Lebenszyklusmodelle organisiert sein.

•••

5.2 Vertragsüberprüfung
•••

5.2.2 Qualitätsrelevante Vertragspunkte

Unter anderem werden folgende Punkte für einen Vertrag oft als zutreffend erachtet:

a) Annahmekriterien
b) Behandlung von Änderungen der Auftraggeberforderungen während der Entwicklung;
c) Behandlung von Problemen, die nach der Annahme entdeckt werden, einschließlich qualitätsbezogener Ansprüche und Auftraggeberbeschwerden;
d Tätigkeiten, die vom Auftraggeber erbracht werden, insbesondere die Rolle des Auftraggebers bei der Festlegung der Forderungen, bei der Installation und bei der Annahme.
e) vom Auftraggeber bereitzustellende Einrichtungen, Werkzeuge und Softwareelemente;
f) anzuwendende Normen und Verfahren
•••

5.3 Spezifikation des Auftraggebers

Für die Durchführung der Softwareentwicklung sollte der Lieferant einen vollständigen und eindeutigen Satz von funktionalen Forderungen haben.
•••

5.4 Planung und Entwicklung
5.5 Planung der Qualitätssicherung
5.6 Design und Implementierung
5.7 Testen und Validieren
5.8 Annahme
5.9 Vervielfältigung, Lieferung und Installierung
5.10 Wartung

6. Qualitätssicherungssystem - ***Unterstützende Tätigkeiten***

6.1 Konfigurationsmanagement
6.2 Lenkung der Dokumente
6.3 Qualitätsaufzeichnungen
6.4 Messungen (am Produkt)
•••
6.9 Schulung

Beispiel eines Qualitätssicherungshandbuchs in [Rot 93]

9.3 Projektorganisation

In der Praxis werden SW-Projekte innerhalb einer Gruppe realisiert. Spezifische Aufgaben innerhalb eines Projekts sind:

- Verwaltungs- /Administrationsarbeit
  - Projektleitung
  - Vertretung des Proj. gegenüber der Firmenleitung
  - Vertretung des Proj. nach außen / Kundenkontakt
  - Termin u. Kostenüberwachung (Controlling)

- Eigentliche Projektarbeit
  - Analyse und Spezifikation
  - Entwurf
  - Implementierung
  - Test und Integration
  - Dokumentation
  - Installation
  - Abnahme

- Qualitätskontrolle / -sicherung

- Konfigurations- / Versionsverwaltung

- Betreuung der Entwicklungsumgebung
  - Pflege / Ausbau / Know How

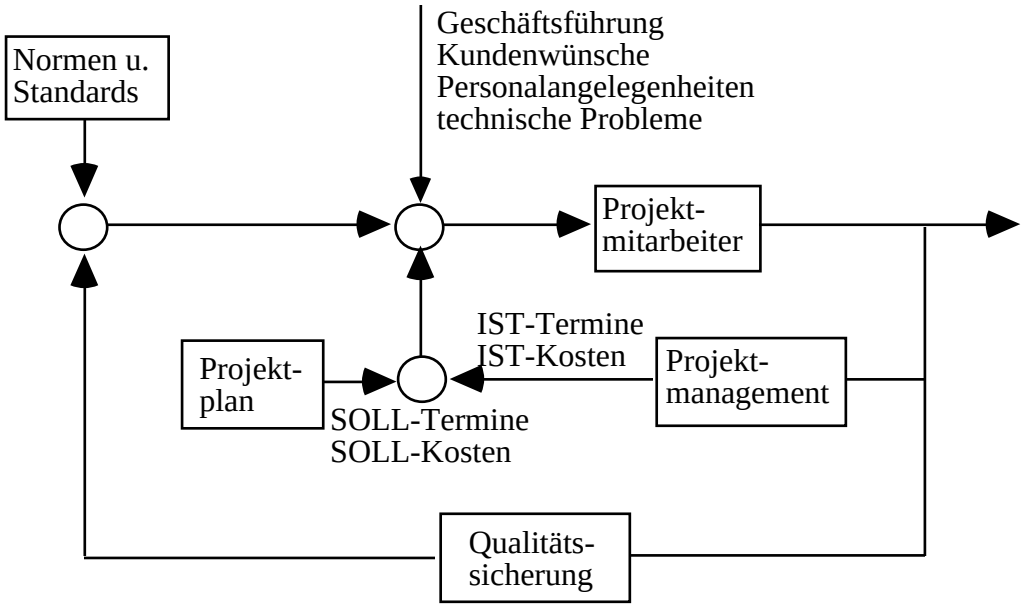
Formen der Projektorganisation sind:

- Hierarchische Projektorganisation
- Einfluß-Organisation
- Matrix-Organisation
- Chef-Programmierer-Team

9.4 Planung, Steuerung und Überwachung des Projekts

Wesentlicher Grund für die Definition eines Vorgangsmodells ist die Möglichkeit einer gezielten Projektverfolgung. Projektverfolgung erfordert:

- exakte Projektplanung
- ständige / regelmäßige IST-Erfassung
- ständiger / regelmäßiger IST-SOLL Abgleich
- möglichst vorausschauende Problemerkennung.



10 Literaturverzeichnis

[Bal 91] Balzert,Helmut:
CASE, Systeme und Werkzeuge, 3. Aufl. BI Wissenschaftsverlag, Mannheim, Wien , Zürich, 1991, (akt. ist die 4. Auflage, 770 S.))

[Bre 92] Breutmann,B.; Burkhardt, R.:

Objektorientierte Systeme: Grundlagen-Werkzeuge-Einsatz; Hanser-Verlag, München, Wien; 1992

[Che] Chen, P.:

Entity-Relationship Model: Toward a Unified View of Data, ACM Transaction on Database Systems, Vol. 1, No. 1

[Coa 91] Coad, P.; Yourdan, E.:

Object Oriented Analysis, 2nd Ed., Prentice-Hall: Englewood Cliffs, NJ,1991

[Coa 91] Coad, P.; Yourdan, E.:

Object Oriented Design, Prentice-Hall: Englewood Cliffs, NJ,1991

[DeM 79] De Marco, T.:

Structured Analisys and System Specification, Prentice hall, Englewood Cliffs, NJ, 1979

[Gei] GEI Software-Tools:

ProMod, Das Ziel ist klar, die Methode treffend; Beschreibung von CASE-Tool Pro-Mod

[HaP 88] Hatley, D.J.; Pirbhai, I.A.:

Stategies for Real-Time System Specification, Dorset House Publ.Co, New York, 1988

[Mey 79] Myers, Glenford J.:

The Art of Software Testing, John Wiley & Sons, 1979, Methodisches Testen von Programmen, dt. Übersetzung von Dr. M. Pieper, 4. Aufl., R.Oldenbourg-Verlag, 1991

[Ott 91] Ott; Hans Jürgen:

Software-Systementwicklung, Praxisorientierte Verfahren und Methoden, Hanser-Verlag,München, Wien, 1991

[Ros 77] Ross, D.T.:

Structured Analysis for requirements definition, IEEE Tansactions on Software Engineering 3, 1/1977, S.6-15

[Ros 77a] Ross, D.T.:

Structured Analysis (SA): A Language for Communicating Ideas, IEEE Tansactions on Software Engineering 3, 1/1977, S.16-34

[Rot 93] Rothery, Brian:

Der Leitfaden zur ISO 9000, mit QM-Musterhandbuch und Erläuterungen, dt. Übersetzung Michael Preising, Hanser-Verlag, München, Wien, 1994

[Sch 90] Schönthaler, F.; Ne´ meth, T.:

Software-Entwicklungswerkzeuge: Methodische Grundlagen, B.G.Teubner,Stuttgart,1990, (350 S.)

[Som 92] Sommerville, Ian:

Software Engineering, 4.ed, Wokingham, Engl. (u.a.), Addison-Wesley, 1992, (649 S.)

[War 85] Ward,P.; Mellor, S.:

Structured Techniques for Real-Time Systems, Yourdon Press/Prentice-Hall, New York, 1985

[Wie 90] Wieken; John-Harry:

Software Produktion; Ziele, Vorgehensweisen, Werkzeuge, Mc Graw-Hill Book Company GmbH, Hamburg, New York, 1990

[You 79] Yourdon, E.; Constantine, L.L.:

Structured Design, Yourdon Press/prentice-Hall, Englewood Cliffs, New York, 1979

[You 89] Yourdon, E.:

Modern Structured Analysis, Yourdon Press/Prentice-Hall, 1989